2-2018

Scheduling in Mapreduce Clusters

Chen He
University of Nebraska-Lincoln, che@cse.unl.edu

SCHEDULING IN MAPREDUCE CLUSTERS

by

Chen He

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professors Ying Lu and David Swanson

Lincoln, Nebraska

February, 2018

SCHEDULING IN MAPREDUCE CLUSTERS

Chen He, Ph.D.

University of Nebraska, 2018

Advisers: Ying Lu and David Swanson

MapReduce is a framework proposed by Google for processing huge amounts of data in a distributed environment. The simplicity of the programming model and the fault-tolerance feature of the framework make it very popular in Big Data processing.

As MapReduce clusters get popular, their scheduling becomes increasingly important. On one hand, many MapReduce applications have high performance requirements, for example, on response time and/or throughput. On the other hand, with the increasing size of MapReduce clusters, the energy-efficient scheduling of MapReduce clusters becomes inevitable. These scheduling challenges, however, have not been systematically studied.

The objective of this dissertation is to provide MapReduce applications with low cost and energy consumption through the development of scheduling theory and algorithms, energy models, and energy-aware resource management. In particular, we will investigate energy-efficient scheduling in hybrid CPU-GPU MapReduce clusters. This research work is expected to have a breakthrough in Big Data processing, particularly in providing green computing to Big Data applications such

as social network analysis, medical care data mining, and financial fraud detection. The tools we propose to develop are expected to increase utilization and reduce energy consumption for MapReduce clusters. In this PhD dissertation, we propose to address the aforementioned challenges by investigating and developing 1) a match-making scheduling algorithm for improving the data locality of Map-Reduce applications, 2) a real-time scheduling algorithm for heterogeneous Map-Reduce clusters, and 3) an energy-efficient scheduler for hybrid CPU-GPU Map-Reduce cluster.

Grant Information

This Ph.D. dissertation is supported by the National Science Foundation Award: 1018467, "CSR: Small: Energy Management for Heterogeneous MapReduce Data Center.

Table of Contents

List of Figures

List of Tables

CHAPTER 1. INTRODUCTION

MapReduce is a framework developed by Google [1] for processing huge amounts of data in distributed computer systems. Hadoop MapReduce [2] is the open source clone of Google's MapReduce. Due to the simplicity of the programming model and the run-time tolerance for node failures, MapReduce is widely used as a platform to solve Big Data problems. In the following part of this dissertation, we will use Hadoop and MapReduce interchangeably.

Big Data was first used in 1970 on atmospheric and oceanic soundings [3]. People use it to refer to a collection of data sets that is too large and complex to be processed by traditional tools. Examples of Big Data include social network logs, financial fraud detections [4,5], AI applications [6,7], and electronic books. The McKinsey Global Institute reports that Big Data will "become a basis of competition, underpinning new waves of productivity growth, innovation, and consumer surplus." With the help of MapReduce, scientists and engineers made significant progresses in many fields. For example, Michael C. Schatz [8] introduced MapReduce to parallelize BLAST that is a DNA sequence alignment program and achieved 250 times speedup. Event logs from Facebook's website are imported into a Hadoop cluster every hour, where they are used for a variety of applications, including analyzing usage patterns to improve site design, detecting spam, data mining and ad optimization [9]. Uber uses MapReduce to analyze mobile trajectory of taxi [10].

As MapReduce clusters get popular, their scheduling becomes increasingly important.

The current MapReduce scheduling, however, has some limitations.

First of all, in a MapReduce cluster, data is distributed to individual nodes and stored in their disks. To execute a map task on a node, we first need to have its input data available on that node. Since transferring data from one node to another takes time and delays task execution, an efficient MapReduce scheduler must avoid unnecessary data transmission. MapReduce default First In First Out (FIFO) scheduler has a policy to improve task data locality. However, it has inevitable deficiencies because of its strict FIFO service policy. Zaharia et al. [9] have developed a delay algorithm to improve the data locality rate. With their technique, a MapReduce scheduler breaks the strict FIFO job order when assigning map tasks to a node. That is, if the first job does not have a map task whose input data is stored in the node's disk (a so-called local task), the scheduler can delay it and assign another job's local map tasks. A maximum delay time D is specified. Only when a job has been delayed for more than D time units will the scheduler assign the job's non-local map tasks. For the delay algorithm, the maximum delay time D is a critical parameter. It is configurable but may need to vary for different workloads and hardware environments.

Secondly, many MapReduce applications [6,7], like online data analytics for spam detection and advertisement optimization, are time sensitive. They require real-time data processing. Scheduling real-time applications in MapReduce environment has become a significant problem. Polo et al. [11] developed a soft real-time scheduler that allows performance-driven management of MapReduce jobs. Dong et al. [12] extended the work by

Polo et al., where a two-level MapReduce scheduler was developed to handle a mixture of soft real-time and non-real-time jobs according to their respective performance demands. Although taking MapReduce jobs' QoS (Quality of Service) into consideration, most existing approaches [11-16] do not provide deadline guarantees for the jobs. Kc and Anyan Wu were the first to investigate the hard real-time scheduling of MapReduce applications [17], where they developed a Deadline Constraint scheduler, aiming to provide time guarantees for MapReduce jobs. However, the Deadline Constraint scheduler has several deficiencies (please see Chapter 5 for details), which may lead to not only resource underutilization but also deadline violations.

Thirdly, with the increasing demands of computational power in big data analysis, Hadoop cluster becomes larger and larger (with thousands of servers) and the cost rises correspondingly. To satisfy the increasing computation power requirement with sustainable costs, General Purpose Graphics Processing Units (GPGPU or simply GPU) are introduced into MapReduce clusters as accelerators. Figure 1.1 provides a performance comparison between CPU and GPU clusters in running the same benchmark. A medium-size hybrid CPU-GPU cluster can be more than 3 times faster than a regular CPU cluster running Hadoop MapReduce applications but with only 1/10 of the hardware costs and 1/20 of the power consumption costs [18]. J. A. Stuart et al. [19,20] built GPMR, an implementation of MapReduce, on a cluster of GPUs. F. Ji et al. [21] developed and optimized another MapReduce framework on GPU by considering GPU multi-level memory hierarchy. K. Shirahata et al. [22] proposed a scheduling technique for hybrid CPU-GPU

Hadoop MapReduce clusters. It tries to minimize the job execution time by using dynamic profiling data of map tasks running on CPU cores and GPU devices. However, they have focused on the performance of MapReduce applications and do not consider the energy consumption costs. Some scientists [22-30] have developed and improved the power model for a hybrid CPU-GPU cluster. But these efforts are not targeted at MapReduce clusters and their models do not consider the specialties of the MapReduce framework.

Last but not least, according to our investigation, energy-efficient real-time scheduling in MapReduce clusters has not been systematically investigated. A. Saifullah et al. [31] developed a method to find intermediate deadlines for synchronous parallel applications running on multi-processor systems, which provides a feasible algorithm, to deal with deadline constraints in MapReduce clusters. However, they do not address energy consumption and data locality issues.

To overcome the aforementioned four limitations, in this PhD dissertation work, we plan to develop:

1. A MapReduce data locality improvement mechanism, which leverages a match-making algorithm to adaptively increase the percentage of local tasks for MapReduce applications.

2. A real-time MapReduce scheduling algorithm that provides a deadline guarantee for real-time MapReduce applications. In this work, we not only enforce the real-time agreement but also maintain good cluster resource utilization.

3. An energy-efficient scheduling algorithm in hybrid CPU-GPU Hadoop clusters, which schedules tasks to available nodes with less energy consumption and high data locality.

The remainder of this dissertation is organized as follows. Chapter 2 presents the background information about Hadoop MapReduce and GPGPU [33]. Related work is described in Chapter 3. Chapter 4 demonstrates the match-making scheduler. In Chapter 5, a real-time scheduler for heterogeneous MapReduce clusters are provided. Chapter 6 includes an energy-efficient scheduler for hybrid CPU-GPU Hadoop clusters. Chapter 7 concludes this dissertation and proposes our future work.

CHAPTER 2. BACKGROUND

Hadoop is mainly composed of two parts: Hadoop Distributed File System (HDFS) [2] and Hadoop MapReduce framework. In this Chapter, we first introduce MapReduce working mechanism, illustrate the MapReduce resource management component: YARN [34-36], and then present HDFS (for latest information about Hadoop community, please refer to [2]). In the end, GPGPU and CUDA [37-39] are described. In the later parts of this dissertation, we will use the terms "Hadoop cluster" and "MapReduce cluster" inter-changeably.

2.1 Hadoop MapReduce

The Hadoop MapReduce structure is illustrated in Figure 2.1:

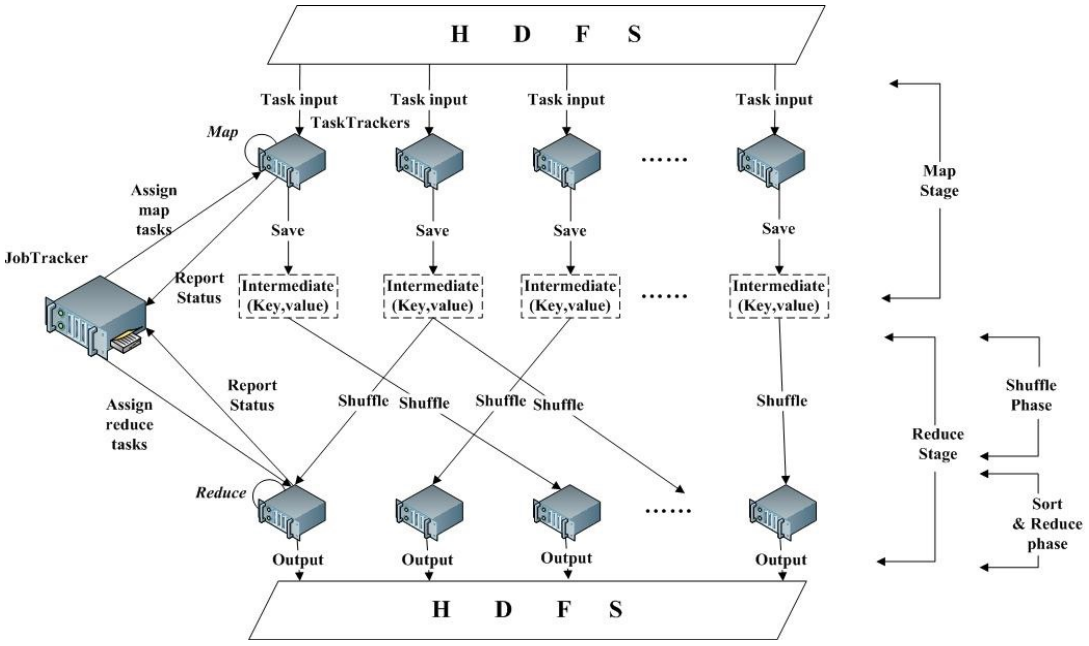


Figure 2.1 MapReduce Framework

A MapReduce cluster is often composed of many commodity PCs, where one PC acts

as the master node and others as slave nodes. A Hadoop cluster uses Hadoop Distributed File System (HDFS) to manage its data. It divides each file into small fixed-size (e.g., 64 MB) blocks and stores several (e.g., 3) copies of each block in local disks of cluster machines. A MapReduce computation is comprised of two stages, map and reduce, which take a set of input key/value pairs and produce a set of output key/value pairs. When a MapReduce job is submitted to the cluster, it is divided into M map tasks and R reduce tasks, where each map task will process one block (e.g., 64 MB) of input data.

A Hadoop cluster uses slave nodes to execute map and reduce tasks. There are limitations on the number of map and reduce tasks that a slave node can accept and execute simultaneously. That is, each slave node has a fixed number of map and reduce slots. Periodically, a slave node sends a heartbeat signal to the master node. Upon receiving a heartbeat from a slave node that has empty map/reduce slots, the master node invokes the MapReduce scheduler to assign tasks to the slave node. A slave node that is assigned a map task reads the content of the corresponding input data block, parses input key/value pairs out of the block, and passes each pair to the user-defined map function. The map function generates intermediate key/value pairs, which are buffered in memory, and periodically written to the local disk and partitioned into R regions by the partitioning function. The locations of these intermediate data are passed back to the master node, which is responsible for forwarding these locations to reduce tasks. A reduce task uses remote procedure calls to read the intermediate data generated by the M map tasks of the job. Each reduce task is responsible for a region (partition) of intermediate data. Thus, it has to re-

trieve its partition of data from all slave nodes that have executed the M map tasks. This process is called shuffle, which involves many-to-many communications among slave nodes. The reduce task then reads in the intermediate data and invokes the reduce function to produce the final output data (i.e., output key/value pairs) for its reduce partition [2].

2.1.1 Task Scheduling & Data Locality

MapReduce framework has a very important feature that is different from traditional distributed computing environments like MPI, OpenMP, and computing Grid, etc. Traditional frameworks move data to where the computation is while MapReduce moves computation to where data is. This way, MapReduce framework gets performance improvement through reduced network traffic. Thus, how to schedule MapReduce jobs becomes an important issue. In the following paragraphs, we will introduce MapReduce scheduling mechanism and its data locality policy.

Hadoop MapReduce framework has a default FIFO scheduler. It schedules MapReduce jobs following a strict FIFO order, i.e., the second job will not be considered if the first job still has a task to be scheduled. Facebook [9] and Yahoo! [36] have developed multi-user schedulers in their production clusters, which will be described in the related work chapter. In the next two paragraphs, we introduce how the FIFO scheduler works and its data locality policy.

Hadoop default FIFO scheduler's data locality policy works as follows. First of all, when a slave node with empty map slots sends the heartbeat signal, the scheduler checks

the first job in the queue. If the job has map tasks whose input data blocks are stored in the slave node, the scheduler assigns the node one of these local tasks. If a slave node has more unused map slots, the scheduler will keep assigning local tasks to the node. However, if the scheduler can no longer find a local task from the first job, it assigns the node one and only one non-local task during this heartbeat interval, no matter how many free slots the node has.

For reduce stage, to evenly distribute reduce tasks to slave nodes, FIFO scheduler only assigns one reduce task to a node in a heartbeat interval because a worker node may be congested if it is assigned many reduce tasks of a job.

2.1.2 Speculative Execution

Since a parallel job's turnaround time is decided by its slowest task, to avoid a MapReduce job from being delayed by the slowest task, MapReduce framework has a speculative execution policy that detects slow tasks and runs a duplicated copy of those tasks.

The MapReduce framework maintains task counters for every job. If a task is 1/3 slower than the average of a job's tasks' execution, the framework will launch another copy of this task on a different slave node. The faster of these two executions will be taken and the other one will be killed. This way, Hadoop MapReduce framework detects the straggler in advance to avoid further delay of execution. There are some researches for speculative execution including LATE [40], SAMR [41], and ESAMR [42], which will be introduced in the related work chapter.

2.1.3 Fault Tolerance

Fault tolerance is an important feature of Hadoop MapReduce. MapReduce clusters do not require sophisticated high-end servers to be used as worker nodes. This assumes that failures exist by default and happen frequently.

Failures are caused by many reasons, for example, network outage, hardware failure, users' misconfiguration, and so on. MapReduce deals with failures through re-execution. Furthermore, Hadoop MapReduce framework has configurable timeout parameters to detect tasks without response. However, some failures cannot be resolved through re-execution. Thus, the maximum-retry-times parameter is used to limit the maximum number of re-executions of a failed task.

For failures caused by an individual slave node, Hadoop MapReduce framework can blacklist a slave node that always fails to execute tasks. In this scenario, the system administrator needs to get involved to restore the blacklisted nodes.

2.1.4 YARN

Since previous Hadoop MapReduce clusters can only schedule MapReduce jobs, the system is not well utilized if users want to run other applications when the MapReduce cluster is not busy. Scientists and system architects proposed the next generation MapReduce framework (YARN) to resolve this problem. In the following paragraphs, we will explain YARN architecture.

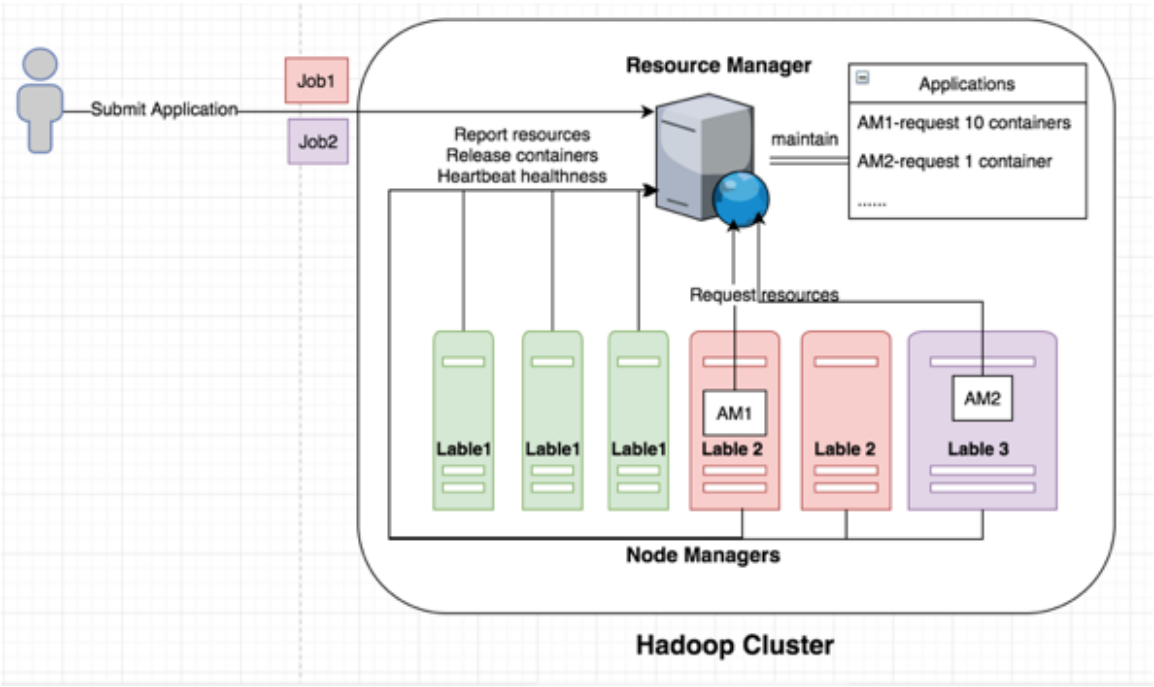


Figure 2.2 YARN Architecture[2]

The basic idea of YARN is to split up the two major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate components. The idea is to have a global ResourceManager (RM), per-node NodeManager (NM), and per-application ApplicationMaster (AM). In YARN, an application is either a single job in the classical sense of a MapReduce job or a job described as a DAG (Directed Acyclic Graph, where a vertex is a processing stage and an edge represents data movement). Users are allowed to submit different types of jobs, create different kinds of AMs, and ask RM for resource allocation.

RM is responsible for allocating resources to the various running applications subject to constraints like capacities, priorities**,** etc. Here, the cluster resources are regarded as a

collection of LXCs (Linux containers) [43]. The "Slot" which is used in an older version of Hadoop MapReduce is not used anymore. The RM's scheduler does not monitor or track application status. Also, it offers no guarantees about restarting failed applications either due to application failure or hardware failures. This scheduler performs its scheduling function based on the resource requirements of that application; which are expressed in terms of resource containers that incorporate elements such as memory, CPU, disk, and network demands. The RM's scheduler has a policy plug-in, which is responsible for partitioning the cluster resources among the various queues, applications etc. The current MapReduce schedulers such as the Capacity Scheduler [44] and the Fair Scheduler [45] would be some examples of the plug-in. The RM and per-node slave, the NodeManager (NM), form the data-computation framework. The RM is the ultimate authority that arbitrates resources among all the applications in the system. The NM is the per-machine framework agent who is responsible for containers, monitoring their resource usage (CPU, memory, disk, network), and reporting to the RM's scheduler. The per-application AM is, in effect, a framework specific library and is tasked with negotiating resources from the RM and working with the NM(s) to execute and monitor the tasks. It is responsible for accepting job-submissions, negotiating the first container for executing the application specific AM and provides the service for restarting the AM container upon a failure. It also has the responsibility of negotiating appropriate resource containers from the RM scheduler.

Now a day, YARN starts to support label scheduling [46] and manages hybrid re-

sources including accelerators such as GPU [47], etc. AM can specify a set of NMs to run its tasks. For example, with label scheduling, an application that requires GPU can run on NMs that have GPU installed.

2.2 Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is an essential component of the Hadoop framework.

HDFS is designed as a highly fault-tolerant, high throughput, and high capacity distributed file system. It is ideal for storing terabytes or even petabytes of data on clusters that may be comprised of commodity hardware. HDFS is based on write-once-read-many and streaming access models. HDFS is very efficient in distributing and storing large amount of data.

2.2.1 HDFS Architecture

HDFS follows the master/slave architecture. The master node in the HDFS cluster is called the Namenode that manages the file system namespace and regulates client accesses to files. There are a number of slave nodes, called Datanodes, which store actual data in units of blocks.

The Namenode maintains a mapping table that maps data blocks to Datanodes in order to process write and read requests from HDFS clients. It is also in charge of file system namespace operations like closing, renaming, and opening files and directories.

The Datanode stores the blocks of files in its local disk and executes the instructions

like replace, create, delete, and replicate from the Namenode. Figure 2.3 (adopted from Apache Hadoop Project) illustrates the HDFS architecture.

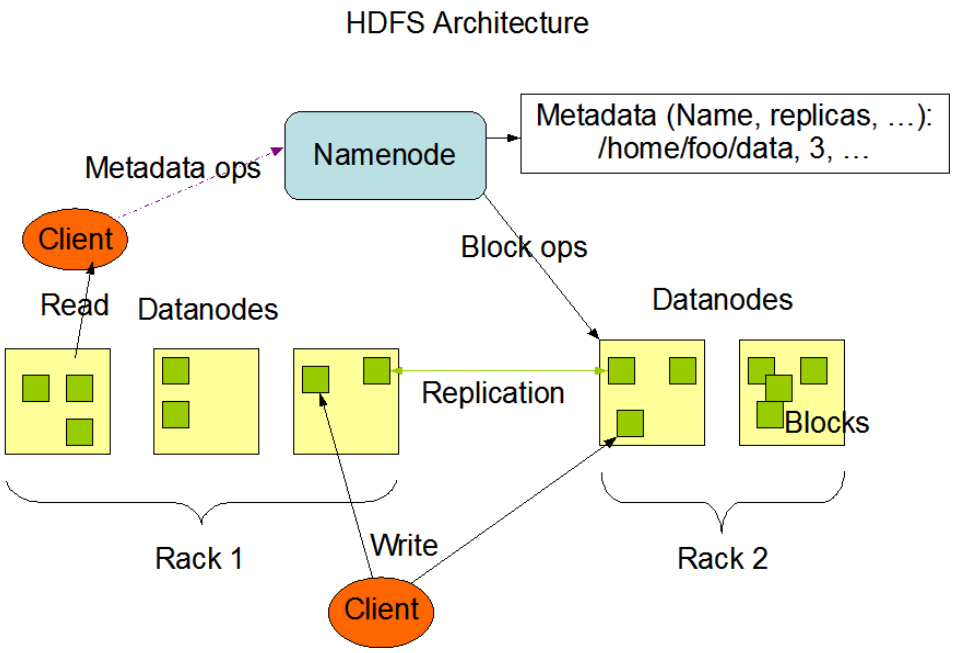


Figure 2.3 HDFS Architecture [2]

A Datanode periodically reports its status (including aliveness, data blocks, etc.) to the Namenode through sending messages (also called heartbeats) and asks the Namenode for instructions. The heartbeat can also help the Namenode to detect connectivity with its Datanodes. Every Datanode maintains an open server socket for data transferring from other Datanodes and user client(s). In order to keep the content of a Namenode in case of failures, HDFS allows a secondary Namenode to periodically backup Namenode data.

2.2.2 Data Placement and Fault-tolerance

HDFS can be deployed on a cluster composed of thousands of nodes. The probability

of failure in a large-scale cluster becomes non-negligible. This means HDFS has to handle the scenario in which some components are non-functional.

HDFS employs an intelligent replica placement policy to guarantee reliability and performance. HDFS keeps 3 replicas for each data block by default. Once a data block is created, the first replica will be placed in a random node. The second replica will be placed in a node that is located in the same rack of that first node. The last replica will be stored in a node from a different rack to guarantee data availability even in the event that an entire rack is down.

2.2.3 Data Balancer

HDFS provides a balancer to equilibrate the disk usage among Datanodes. When placing data blocks, the Namenode randomly picks a node to place the first copy of a data block. This mechanism may result in some nodes with smaller capacity having higher percentage of disk usage. The balancer is designed to solve this problem. It allows an administrator to balance HDFS Datanodes based on disk usage percentage.

2.4 GPGPU and CUDA

"General-Purpose Graphics Processing Unit (GPGPU) is utilizing the graphics-processing unit (GPU) to do computation for applications that are traditionally handled by the CPU" [33]. It is widely used in supercomputers as an accelerator to enhance the computational power. The comparison between CPU and GPU is detailed documented [33, 37-39].

CUDA [37] (Compute Unified Device Architecture) is a parallel computing architecture designed for GPUs and proposed by NVIDIA in 2006. It enables programmers to write C (C-CUDA) code to utilize GPUs for processing non-graphical data. C-CUDA programs are compiled using a specialized Path Scale Open64 C compiler. CUDA has been widely used to accelerate computations which otherwise take much longer or are intractable with the current technology, e.g., molecular dynamics simulation, electronic design automation, accelerated rendering of 3D graphics, speech indexing, and physical simulations.

With a design principle different from traditional CPUs, GPUs are based on a parallel throughput architecture that is aimed at executing a large number of concurrent threads slowly, as opposed to executing a single thread very fast. CUDA provides APIs for multiple operating systems, including Windows, Linux, and recently Mac OS X. Moreover, CUDA is supported by all GPUs recently designed and manufactured by NVIDIA [48], i.e., from the G8X series onwards, including GeForce, Quadro and the Tesla product lines. NVIDIA maintains compatibility among different generations of their GPUs such that CUDA programs developed for the GeForce 8 series will also work without modification on all future NVIDIA graphics cards.

With a radically different design, CUDA is superior over traditional GPGPU solutions with graphics APIs. For example, CUDA supports Scattered Reads, i.e., programs can access memory at arbitrary addresses on both the host and the device. Moreover, CUDA has a solid hardware implementation of floating-point arithmetic, which is essential for

scientific computations.

Admittedly, CUDA also suffers several drawbacks at the current stage. For instance, C-CUDA disallows the uses of recursion and function pointers, which might place a burden on programmers while developing CUDA programs in some scenarios. Although equipped with very fast internal cache memories, GPU might suffer from the limited bus bandwidth along the data-path to the CPU. Furthermore, the deep memory hierarchy and intricate internal mechanisms might have huge performance implications if CUDA programs are written without accounting for such complexities in the design. Nevertheless, we believe the advantages of massive-parallelization offered by CUDA surely outweigh the drawbacks, as mentioned above, in real world applications.

Besides C, CUDA has bindings for most mainstream programming languages, including C++, Java, .NET, Perl, Python, Ruby, Lua, FORTRAN, and Matlab. In this work, we focus on JCuda [48], which is the CUDA binding for the Java language, which is being actively developed with support for the most recent CUDA API. JCuda provides a solid foundation for using CUDA libraries in Java applications.

We use a very simple array summation example in Figure 2.4 to demonstrate how GPU and CPU cooperate together. In order to distinguish arrays in main memory from those in GPU's global memory, we use "dev" (short for device) plus capital characters to identify three arrays in GPU's global memory. First of all, CPU allocates three arrays in the main memory, array "a" and "b" contains elements we want to sum where array "c" is used to store the results (step 1). Correspondingly, CPU also needs to allocate three arrays

in GPU's global memory that is the bottom rectangle in GPU (step 2). CPU copies array "a" and "b" contents from main memory into GPU's global memory (step 3). On the GPU side, the first 4 rows of rectangles from top are computation elements and the fifth row of rectangles are shared memories. Communication between shared memories should employ global memory. The computation element needs to load array "devA" and "devB" into shared memories before launching the summation (step 4).

After the summation operation (step 5), array "devC" will be stored to global memory from shared memory (step 6). The next step is to copy array "devC" to array "c" from global memory to main memory (step 7). Finally, all memory space in shared memories and global memory will be recycled (step 8).

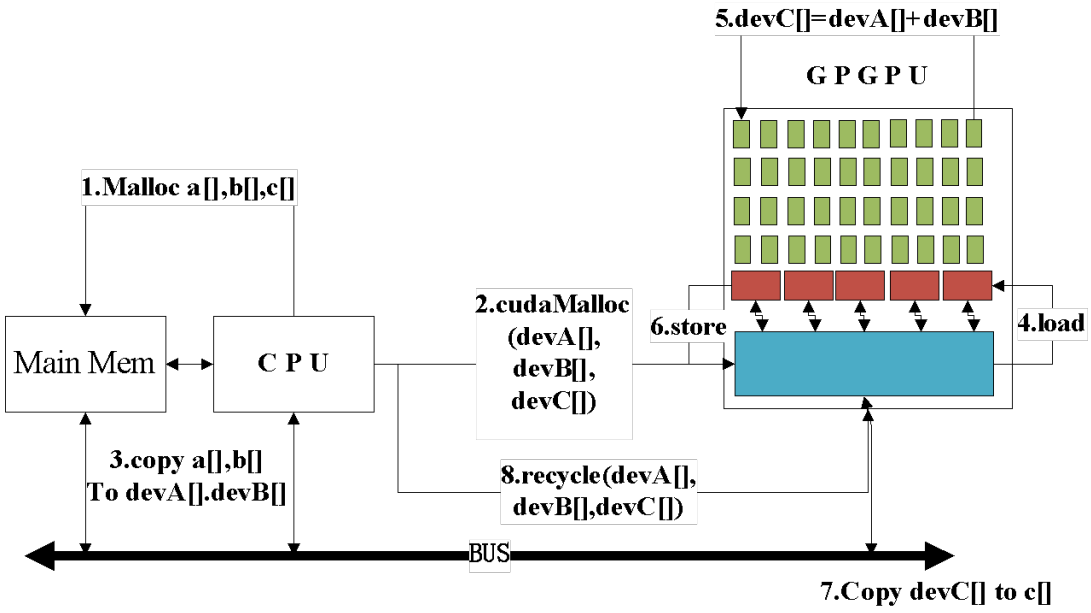


Figure 2.4 CPU + GPU Architecture

Comparing with CPU, GPGPU is more energy efficient [50]. According to NVIDIA's

research reports: "On the Top500 Supercomputers list — a biannual ranking of supercomputing sites around the world — the number of GPU-powered systems is rapidly growing. Today, three of the five fastest supercomputers in the world are NVIDIA GPU-powered. And these systems are much more energy efficient." More and more Hadoop clusters are equipped with GPUs to accelerate computational intensive MapReduce applications [20-33]. In this dissertation, we will develop an energy efficient scheduler for Hadoop MapReduce clusters that have GPUs.

CHAPTER 3. RELATED WORK

MapReduce framework was first proposed by Google [1]. It is a fault-tolerant platform used for parallel processing huge amounts of data. Hadoop is a well-accepted open source implementation of Google's MapReduce framework. In this dissertation, we focus on research work related to Hadoop MapReduce [2] from two aspects: scheduling and power management.

3.1 MapReduce Scheduling

Early versions of Hadoop had a very simple approach to scheduling users' jobs: they ran in order of submission, using a FIFO scheduler by default. Typically, each job would use the whole cluster, so jobs had to wait their turn. As MapReduce clusters got popular, their scheduling became increasingly important. However, the default FIFO scheduler does not support many desired features like QoS guarantee, resource sharing, preemption, etc. Then, scientists started to explore various algorithms to improve MapReduce scheduling [15-62]. In this section, we mainly focus on introducing research work from following areas that are related to my dissertation: detecting speculative tasks, improving data locality, providing QoS guarantee, and scheduling MapReduce jobs in hybrid (CPU-GPU) clusters.

Hadoop's scheduler implicitly assumes that cluster nodes are homogeneous and MapReduce tasks make progress linearly and uses these assumptions to decide when to speculatively re-execute tasks that appear to be stragglers [2]. To overcome this limitation and

make the speculative execution mechanism effective in heterogeneous environments, researchers then developed LATE (Longest Approximate Time to End) scheduler [40], SAMR (Self-Adaptive MapReduce Scheduling) algorithm [41], and ESAMR algorithm [42].

MapReduce framework has a significant difference from previous parallel processing platforms, like computing Grid [63]. Previous frameworks move data to where the computation resource is located. However, MapReduce allocates the computation to where the data is stored. That is, when scheduling a task, MapReduce system will first consider a server that stores this task's input data in local disk. To enhance this data locality in executing MapReduce application, researchers have used technologies like prefetching [80], node status prediction [81], and delay scheduling algorithm [40].

In order to improve MapReduce cluster utilization, researchers introduce resource sharing [64-66], iterative execution [67-70], load balancing [72], online aggregation [73], genetic algorithm based data-aware group scheduling [74], introducing erasure coding in storage [75], network-aware task placement scheduling [76], and multi-object scheduling [77] into MapReduce. Yahoo! developed a multi-queue scheduler called Capacity Scheduler [44] for Hadoop clusters, where every queue is guaranteed a fraction of the capacity. Within a queue, it supports job priorities but no job preemption is allowed. To prevent one or more users from occupying all resources of a queue, each queue enforces a limit on the percentage of resources allocated to a user at any given time if there is competition for resources.

The fair scheduler [40] also supports multiple queues (also called pools). Jobs are organized into pools and resources are fairly divided between these pools. By default, there is a separate pool for each user so that each user gets an equal share of the cluster. Within each pool, jobs can be scheduled using either fair sharing or FIFO scheduling. Fair sharing scheduling is a method of assigning resources to jobs such that all jobs get, on average, an equal share of resources over time. When there is a single job running, that job uses the entire cluster. When other jobs are submitted, task slots that free up are assigned to the new jobs so that each job gets roughly the same amount of CPU time. Unlike the default Hadoop FIFO scheduler, which forms a queue of jobs based on job arrival times, fair sharing scheduling mechanism guarantees that short jobs finish in reasonable time without starving long jobs. It provides an easy way to share a cluster between multiple users [40].

Since many MapReduce applications [73], including online data analytics for spam detection and ad optimization, require real-time data processing, scheduling real-time applications in MapReduce Environment become an important problem [33,68-73]. Scientists have already established many important theories for real-time scheduling [90-98], especially in distributed systems [99-115]. For MapReduce real-time scheduling, J. Polo et al. [68] developed a scheduler that focuses on MapReduce jobs that have soft deadlines. It estimates jobs' execution times and tries to let jobs satisfy their deadlines by scheduling resources according to the estimated finishing times. Dong et al. [70] extended the work by Polo et al., where a two-level MapReduce scheduler was developed to

schedule mixed soft real-time and non-real-time jobs according to their respective performance demands. Linh T.X. Phan et al [72] built HadoopRT that focuses on enhancement of EDF with locality-awareness and overload handling in cloud environment. They defined a parameter to describe the execution time difference between local and non-local tasks. HadoopRT can adjust its scheduling policies according to this parameter to improve MapReduce applications' performance. Chen F. et al. proposed a system that schedule real-time MapReduce applications based on job size [115]. However, they did not consider energy consumption and hybrid clusters.

Kamal Kc et al. [69] developed a scheduler for MapReduce applications with hard deadlines. It also estimates the job finishing time according to available resources in a MapReduce cluster. If a job cannot finish before the hard deadline, the scheduler will not execute the job and will instead inform the user to adjust the job deadline. However, it has deficiencies that may cause deadline misses and low hardware utilization.

After YARN was created, scientists and architects started to improve its performance by optimizing the scheduling algorithms. Yao et al. [119] proposed YARN scheduler, named HaSTE, which can effectively reduce the make-span of MapReduce jobs in YARN by leveraging the information of requested resources, resource capacities, and dependency between tasks. Lin et al. [120] employed real time ABS-YARN, which is a formal language for executable modeling of deployed virtualized software, to optimize the deployment decision in the cloud to reduce scheduling cost.

In recent years, accelerators and heterogeneous architectures, especially GPUs, have

emerged as major players in high performance computing [119-133]. For some types of MapReduce applications that require significant amount of computation (like machine learning and data mining algorithms), hybrid CPU-GPU architecture can be a high- performance, scalable, cost-effective, and power-efficient solution. [133-142].

B. Catanzaro et al. [135] created a platform that can automatically generate GPU CUDA code for MapReduce applications. J.A. Stuart et al. [20] also created a MapReduce framework on a cluster of GPUs to do volume rendering. Chen et al. [140] optimized MapReduce performance for GPU through reduction-based method that allows MapReduce to carry out reductions in shared memory. They designed and implemented their MapReduce framework in a single AMD Fusion chip. Qiao Z. et al. [138] built MR-Graph, an implementation of MapReduce, on a cluster of GPUs. F. Ji et al. [139] developed and optimized performance for another MapReduce framework on GPU by considering GPU multi-level memory hierarchy. However, all aforementioned frameworks did not focus on Hadoop MapReduce that is a widely accepted MapReduce platform. Liu LF. Et al. [142] developed an adaptive MapReduce framework for GPUs

K. Shirahata et al. [22] proposed a scheduling technique in hybrid CPU-GPU Hadoop MapReduce clusters, which minimizes job execution time via dynamic profiling of Map tasks running on CPU cores and GPU devices. However, they focused on optimizing the map stage of MapReduce applications and did not consider the energy consumption in their scheduling algorithm.

3.2 Power Management in Hadoop Cluster

With the increasing scale of MapReduce clusters, the cost of maintaining a MapReduce cluster becomes larger and larger. How to reduce MapReduce cluster power bill turns out to be a critical concern. Scientists have done some research work on power management of MapReduce clusters [144-148] inspired by previous theories in cluster power management.

Lang et al. [143] provided an algorithm that only keeps the smallest number of servers that can guarantee data integrity in HDFS. However, it is not flexible if the cluster executes time-sensitive online applications. T. Wirtz et al. [144] used an experimental approach to study the scalability of performance, energy, and efficiency of MapReduce for computation intensive workloads. They proposed a power management policy through resource allocation that changes the number of available workers and DVFS (Dynamic Voltage and Frequency Scaling) that adjusts the processor frequency based on current computational needs. M. Cardosa et al. [145] considered power management in cloud environment through VMs management algorithm. Jerry Chou et al. [146] built an algorithm that can monitor system utilization and re-direct requests to existing powered-on recourses. N. Yigitbasi et al. [147] investigated scheduling algorithm in a heterogeneous cluster made of high performance nodes and low power nodes. His scheduling algorithm is limited in this specific hardware environment. Chen et al. [148] presented BEEMR through tracing MapReduce interactive analytics in Facebook Hadoop production cluster. It first categorizes MapReduce applications into different job zones according to service

types, for example, batching jobs, interactive jobs, etc. Then, BEEMR saves energy by flexibly adjusting the number of servers that work for interactive jobs according to system requests. However, it is based on Facebook's workload that is mainly composed of online queries. It may not be a good fit for MapReduce clusters that work in other industries like banks, health care companies, etc.

Since hybrid CPU-GPU cluster becomes more and more popular, scientists start to consider how to predict hybrid CPU-GPU cluster power consumption. Ren DQ et al. [149] proposed an empirical power model for GPU to predict the optimal number of active processors (CPU and GPU) for a given application. W. Liu et al. created a waterfall model [150] which uses a mapping algorithm to apply different energy saving strategies to keep the system at lower energy levels. In their mapping algorithm, they adopted dynamic voltage scaling, dynamic resource scaling and -migration for GPU to reduce energy consumption. H. Huo et al. [151] proposed a flexible energy efficient task-scheduling scheme for heterogeneous tasks in the heterogeneous GPU-enhanced clusters. It includes a system model to describe hardware heterogeneity and a task model to characterize application heterogeneity in a cluster. However, they provide no evaluation data from either simulation or real-system experiment. Kim et al. [153] proposed an algorithm about power management in MapReduce hybrid cluster. But they did not consider data locality.

In summary, the previous research works outlined here do not consider multiple constraints including energy efficiency, data locality, and throughput together in the hybrid heterogeneous Hadoop clusters. Instead, this dissertation will focus on resolve this hard

problem step by step.

CHAPTER 4. MATCHMAKING SCHEDULER

In a MapReduce cluster, data are distributed to individual nodes and stored in their disks. To execute a map task on a node, we need to first have its input data available on that node. Since transferring data from one node to another takes time, delays task execution, and consumes extra energy. An efficient MapReduce scheduler must avoid unnecessary data transmission.

We will focus on the problem of decreasing data transmission in a MapReduce cluster and we develop a scheduling technique to improve map tasks' data locality rate. For a given execution of MapReduce workload, the data locality ratial is defined in this dissertation as the ratio between the numbers of local map tasks and all map tasks, where a local map task refers to a task that has been executed on a node that contains its input data. A low data locality rate means more data transfer between machines and higher network traffic. To avoid unnecessary data transfer, our scheduling technique aims to achieve high data locality rate and also short response time for MapReduce clusters. We developed a new technique to enhance the data locality. The main idea of the technique is as follows. To assign tasks to a node, local map tasks are always preferred over non-local map tasks, no matter which job a task belongs to, and a locality marker is used to mark nodes and to ensure each node a fair chance to grab its local tasks. Experiments are carried out to evaluate the aforementioned techniques and experimental results show that our technique leads to the high data locality rate and the low response time for map tasks. Unlike the delay algorithm [40], our technique does not require the tuning of the delay parameter.

4.1 Hadoop Default FIFO Scheduler

The Hadoop default FIFO scheduler has already taken data locality into account. When a slave node with empty map slots sends the heartbeat signal, the MapReduce scheduler checks the first job in the queue. If the job has map tasks whose input data blocks are stored in the slave node, the scheduler assigns the node one of these local tasks. If a slave node has more unused map slots, the scheduler will keep assigning local tasks to the node. However, if the scheduler can no longer find a local task from the first job, it assigns the node one and only one non-local task during this heartbeat interval, no matter how many free slots the node has.

This default FIFO scheduler, however, has deficiencies. First of all, it follows the strict FIFO job order to assign tasks, which means it will not allocate any task from other jobs if the first job in the queue still has an unassigned map task. This scheduling rule has a negative effect on the data locality because another job's local tasks cannot be assigned to the slave node unless the first job has all its map tasks (many of which are non-local to the node) scheduled.

Secondly, the data locality is randomly decided by the heartbeat sequence of slave nodes. If we have a large cluster that executes many small jobs, the data locality rate could be quite low. As mentioned, in a MapReduce cluster, tasks are assigned to a slave node in response to the node's heartbeat. With the FIFO scheduler, heartbeats are also processed in a FIFO order and a node is assigned a non-local map task when there is no local task from the first job. In a large cluster many nodes heartbeat simultaneously.

However, a small job has less input data that are stored in a small number of nodes. It is thus a high probability event that the scheduler assigns tasks to slave nodes that do not have the small job's input data but give heartbeats first. For example, if we execute a job of 5 map tasks on a MapReduce cluster of 100 slave nodes, it is unlikely to get a high locality rate. Since each map task needs one input data block, which by default has 3 replicas stored in 3 nodes, at most 15 out of 100 nodes have input data for the job, i.e., the job's tasks are all non-local to at least 85 nodes. A slave node with empty map slots that sends in a heartbeat first will always be assigned at least one map task, local or non-local. It is highly likely that the job's tasks will be assigned to many of those 85 nodes which do not have the input data blocks before a node even gets a chance to grab a local task from the job.

4.2 Delay Scheduling Algorithm

Zaharia et al. [40] have developed a delay scheduling algorithm to improve the data locality rate of Hadoop clusters. It relaxes the strict job order for task assignment and delays a job's execution if the job has no map task local to the current slave node. To assign tasks to a slave node, the delay algorithm starts the search at the first job in the queue for a local task. If not successful, the scheduler delays the job's execution and searches for a local task from succeeding jobs. A maximum delay time D is set. If a job has been skipped long enough, i.e., longer than D time units, its non-local tasks will then be assigned for execution. With the delay scheduling algorithm, a job's execution is postponed to wait for a slave node that contains the job's input data. Here, the delay time D is a key

parameter. By default, it is set at 1.5 times the slave node's heartbeat interval. However, to obtain the best performance for the delay scheduling algorithm, we have to choose an appropriate *D* value. If the value is set too large, job starvations may occur and affect performance. On the contrary, a too small *D* value allows non-local tasks to be assigned too fast. For different kinds of workloads and hardware environments, the best delay time may vary. To get an optimal delay time always requires careful *D* value tuning.

In addition, this delay algorithm allows a node to obtain multiple non-local map tasks in a heartbeat interval if the node has more than one free slot. In some situations, this algorithm could perform worse than the FIFO scheduler's locality enhancement policy because the latter only allows one non-local task to be assigned to a node in a heartbeat interval.

Although first developed to improve the data locality of the Hadoop fair scheduler [20], delay scheduling is applicable beyond fair sharing, in general, applicable to any scheduling policy (e.g., FIFO) that defines an order in which jobs should be given resources [2]. It is very popular and widely used in Hadoop clusters.

4.3 Matchmaking Scheduling Algorithm

This section presents our new technique for enhancing the data locality in MapReduce clusters. The main idea behind our technique is to give every slave node a fair chance to grab local tasks before any non-local tasks are assigned to any slave node. Since our algorithm tries to find a match, i.e., a slave node that contains the input data, for every unassigned map task, we call our new technique the matchmaking scheduling algorithm.

First of all, like the delay scheduling algorithm, our matchmaking algorithm also relaxes the strict job order for task assignment. If a local map task cannot be found in the first job, the scheduler will continue searching the succeeding jobs. Second, in order to give every slave node a fair chance to grab its local tasks, when a node fails to find a local task in the queue for the first time in a row, no non-local task will be assigned to the node. That is, the node gets no map task for this heartbeat interval. Since during a heartbeat interval, all slave nodes with free map slots have likely given their heartbeats and been considered for local task assignment, when a node fails to find a local task for the second time in a row (i.e., still no local task a heartbeat interval later), to avoid wasting computing resources, the matchmaking algorithm will assign the node a non-local task. This way, our algorithm achieves not only high data locality rate but also high cluster utilization. To enforce the aforementioned rule, our algorithm gives every slave node a locality marker to mark its status. If none of the jobs in the queue has a map task local to a slave node, depending on this node's marked value, the matchmaking algorithm will decide whether or not to assign the node a non-local task. Third, our matchmaking algorithm allows a slave node to take at most one non-local task every heartbeat interval. At last, all slave nodes' locality markers will be cleared when a new job is added to the job queue. Because a new job may comprise new local tasks for some slave nodes, upon the new job's arrival, our algorithm resets the status of all nodes and again starts the all-to-all task-to-node matchmaking process. Tables 4.1 and 4.2 give the pseudo code of our algorithm. Like delay scheduling algorithm, our matchmaking algorithm is applicable to any

scheduling policy (e.g., FIFO or fair sharing scheduling) that defines an order in which jobs should be given resources.

Table 4.1 Matchmaking Algorithm

Algorithm 1: Matchmaking Scheduling Algorithm

```
1:  for each node i of the N slave nodes do
2:      set LocalityMarker[i]=null
3:  end for
4:  //Upon receiving a heartbat from node i:
5:   while node i has free slots, i.e., its free slot count s>0
6:      set previousMarker=LocalityMarker[i]
7:      for each job j in the JobQueue do
8:          if  job j has an unassigned local task t then
9:              assign t to node i
10:             set s=s-1
11:             if LocalityMarker[i]==null then
12:                 LocalityMarker[i]=1
13:             else LocalityMarker[i]+=1
14:             end if
15:             break for
16:             else continue
17:         end if
18:     end for
19:     if previousMarker==LocalityMarker[i] then
20:         set LocalityMarker[i]=0          //mark this node
21:         break while
22:     else if LocalityMarker[i]==0 then
23:         assign node i a non-local task t' from the first job in the JobQueue
24:         set s=s-1
25:         break while
26:     end if
27: end while
```

Table 4.2 Locality Marker Maintenance

Algorithm 2: Locality Marker Cleaning Algorithm
1: //When a new job *j* is added into the *JobQueue*: 2: for each node *i* of the *N* slave nodes do 3: set *LocalityMarker[i]=null* 4: end for

4.4 Evaluation of Different Data Locality Policies

To evaluate our matchmaking scheduling algorithm, we compare it with the Hadoop default FIFO scheduler and the delay scheduling algorithm. Two metrics, i.e., map tasks' *data locality ratio* and *average response time*, are used for evaluation.

We run experiments in a private cluster of 1 head node and 30 slave nodes that are configured as one rack. We modify Hadoop and integrate our matchmaking algorithm with both Hadoop FIFO scheduler and Hadoop fair scheduler. The cluster is configured with a block size of 128MB, which follows Facebook's Hadoop cluster block size configuration [20]. Table 4.3 lists our Hadoop cluster hardware environment and configuration.

Table 4.3 Experimental Environment

Nodes	Quantity	Hardware and Hadoop Configuration
Master node	1	2 single-core 2.2GHz Optron-64 CPUs, 8GB RAM, 1Gbps Ethernet
Slave nodes	30	2 single-core 2.2GHz Optron-64 CPUs, 4GB RAM, 1 Gbps Ethernet, 1 rack, 2 map and 1 reduce slots per node

4.4.1 Experimental Environment

To evaluate our matchmaking algorithm, we create a submission schedule that is simi-

lar to the one used by Zaharia et al[20]. They generated a submission schedule for 100 jobs by sampling job inter-arrival times and input sizes from the distribution seen at Facebook over a week. By sampling job inter-arrival times at random from the Facebook trace, they found that the distribution of inter-arrival times was roughly exponential with a mean of 14 seconds.

They also generated job input sizes based on the Facebook workload, by looking at the distribution of the number of map tasks per job at Facebook and creating datasets with the correct sizes (because there is one map task per 128 MB input block). Job sizes were quantized into nine bins, listed in Table 4.4 [20], to make it possible to compare jobs in the same bin within and across experiments. Our submission schedule has similar job sizes and job inter-arrival times. In particular, our job size distribution follows the first six bins of job sizes shown in Table 3.4, which cover about 89% of the jobs at the Facebook production cluster. Because most jobs at Facebook are small and our test cluster is limited in size, we exclude those jobs with more than 300 map tasks. Like the schedule in [20], the distribution of inter-arrival times is exponential with a mean of 14 seconds, making our submission schedule totally 21 minutes long.

We generate 100 input data blocks in Hadoop Distributed File System (HDFS). The popularity of blocks is assumed to follow a uniform distribution. That is, when a job requests a block, it is evenly likely to be any one of the blocks stored in HDFS. Each of the blocks has 2 replicas. We distribute and store these 200 block replicas evenly in 30 slave nodes, ensuring no two replicas of a block be stored in the same node. As a result, every

slave node contains about 6 (or 7) blocks. By uniformly distributing blocks among our cluster nodes, we avoid hotspots of data requests.

We use our submission schedule for two application workloads. One is *loadgen* that is a test example from the Hadoop test package. It loads input data and outputs a fraction of the data intact. This application has been used as a test workload for the delay algorithm [20]. The other application we adopt is *wordcount* that is a classic example of Hadoop applications.

As mentioned, we have modified Hadoop and integrated our matchmaking algorithm with both Hadoop FIFO scheduler and Hadoop fair scheduler.

In our experiments, we always configure the cluster to have just one job queue. With Hadoop fair scheduler, all jobs in a queue are scheduled following either fair sharing or FIFO scheduling rule. With fair sharing scheduling, resources are assigned to jobs such that all jobs get, on average, an equal share of resources over time. We have tested the performance of delay algorithm within Hadoop fair scheduler. Depending on the applied scheduling rules (FIFO or fair sharing), we have two different versions: FIFO with delay algorithm and Fair with delay algorithm. Since we have tested our matchmaking algorithm within Hadoop FIFO scheduler, when testing matchmaking algorithm within Hadoop fair scheduler, only the fair sharing scheduling rule is applied.

We thus run each workload under five schedulers: Hadoop FIFO scheduler, Hadoop FIFO scheduler with matchmaking algorithm, FIFO with delay algorithm, Fair with delay algorithm, and Fair with matchmaking algorithm.

For the delay algorithm, we need to configure the maximum delay time *D*. In our experiments, a total of 8 different *D* values are chosen. They are from 0.1 to 10 times the slave node's heartbeat interval. Since we configure the heartbeat interval to be 3 seconds long, the maximum delay time *D* changes from 0.3 to 30 seconds.

To eliminate the possible randomness of cluster hardware status, every point shown in the figures is the average of three runs.

Table 4.4 Facebook Workload

Bin	#Maps	%Jobs at Facebook	#Maps in Benchmark	# of jobs in Benchmark
1	1	39%	1	38
2	2	16%	2	16
3	3-20	14%	10	14
4	21-60	9%	50	8
5	61-150	6%	100	6
6	151-300	6%	200	6
7	301-500	4%	400	4
8	501-1500	4%	800	4
9	>1501	3%	4800	4

4.4.2 Experiments

We first use the data locality rate to measure the performance of the following three schedulers: Hadoop FIFO scheduler, Hadoop FIFO scheduler with matchmaking algorithm, and FIFO with delay algorithm. Given a workload execution, the data locality rate is defined as,

$$\text{Data Locality Rate} = \frac{l}{n} \qquad (4.1)$$

where l is the number of local map tasks and n is the total number of map tasks. To make the figures properly fits the page, we did not follow numerical scale of delay times in x coordinate but simply listed them side by side to show the trend of data locality rate when delay time increases.

Our experimental results on data locality rate with the two application workloads are shown in Figures 4.1 and 4.2. As we can see, the data locality rate achieved with the delay algorithm increases with the maximum delay time D. The longer a job is delayed, the higher the probability that the job finds slave nodes that contains the input data blocks. In following diagrams, we use MM to represent Matchmaking algorithm.

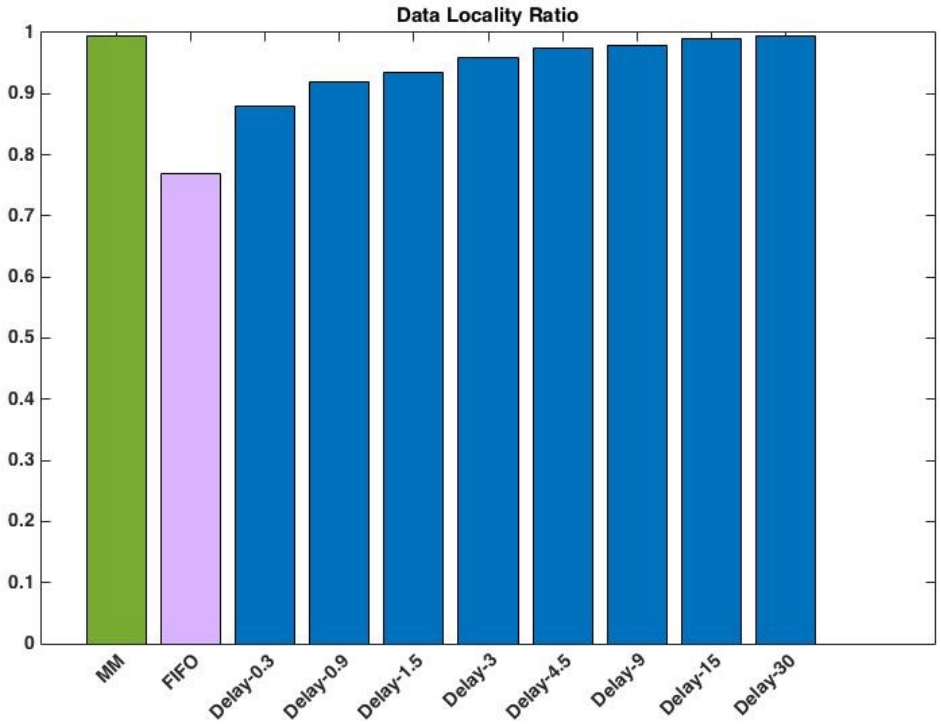


Figure 4.1 Loadgen Workload: Data Locality Ratio

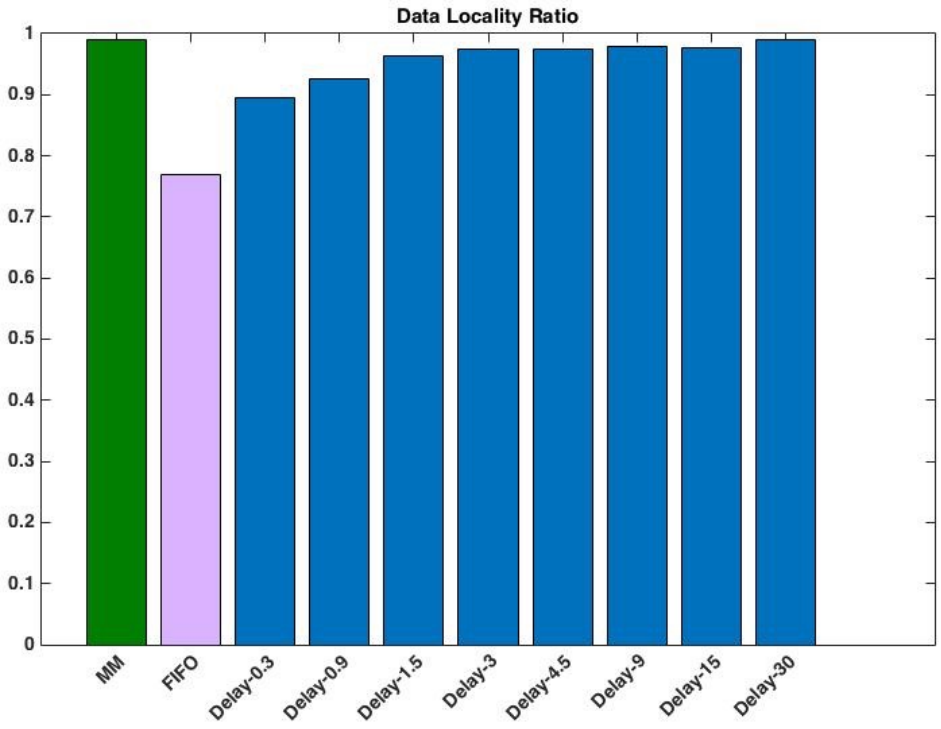


Figure 4.2 Wordcount Workload: Data Locality Ratio

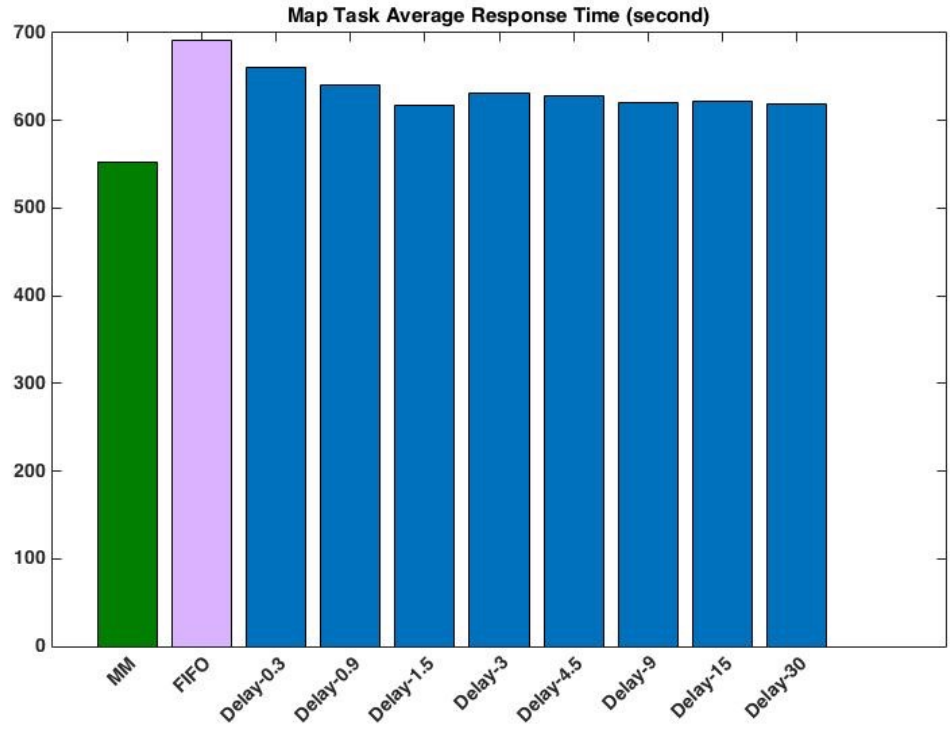


Figure 4.3 Loadgen Workload: Map Tasks' Average Response Time

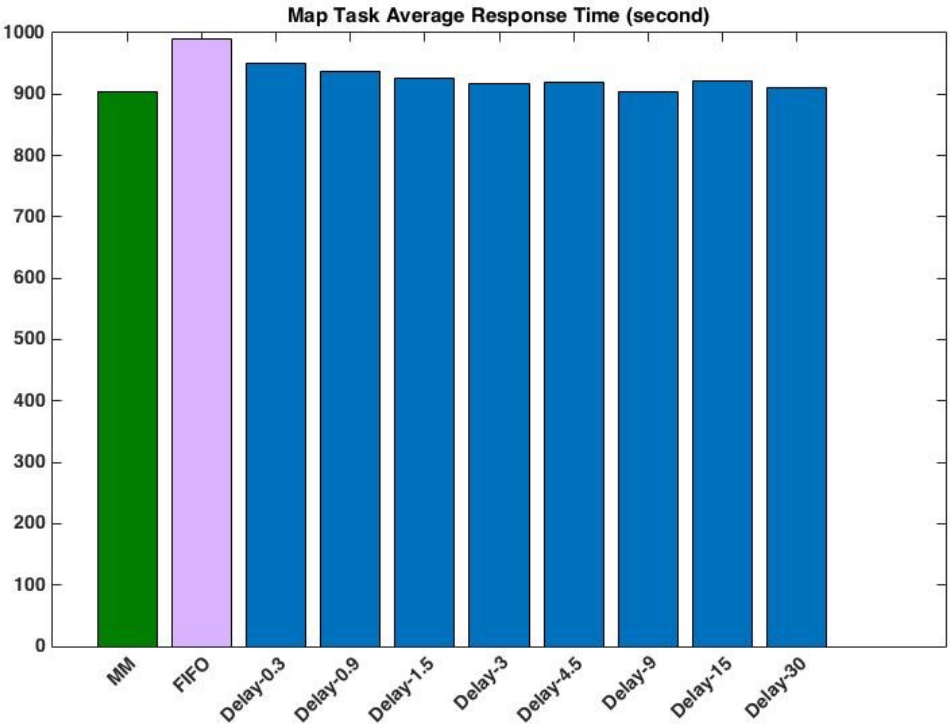


Figure 4.4 Wordcount Workload: MapTasks' Average Response Time

Figures 4.1 and 4.2 also show that the FIFO scheduler leads to the worst performance, i.e., the lowest data locality rate. However, when we integrate our matchmaking technique with the FIFO scheduler, the algorithm achieves the highest data locality rate, better than any of those achieved with the delay algorithm of different D values.

To evaluate the algorithms' performance only via the data locality rate is not enough since we can easily design an algorithm that enforces the constraint that all tasks have to be executed on slave nodes that contain their input data, leading to 100% data locality rate but also long response time for map tasks due to the long delay required to satisfy the strict constraint. Therefore, we also evaluate our algorithms by another metric: the average response time of all map tasks. Figures 4.3 and 4.4 present the experimental results. As shown in the figures, when we run the workloads with the FIFO scheduler, we get the longest average response time for map tasks. After enhancing the FIFO scheduler with our matchmaking algorithm, we reduce the average response time significantly.

For the delay algorithm, although the higher the D value, the better the data locality rate (see Figures 4.1 and 4.2), the relationship between the D value and the average response time is not so straightforward. When running the loadgen workload, the average response time varies with the D value, e.g., getting smaller when D increases from 0.3 to 1.5 seconds but longer when D increases from 1.5 to 3 seconds (see Figure 4.3). The lowest average response time is achieved when the maximum delay time is set at 30 seconds (see Figures 4.1 & 4.3-loadgen). But, that is not the optimal D value when running the

wordcount workload. As shown in Figure 4.2 (and also in Figure 4.4-wordcount), when D = 9 or 15 seconds, we get the best average response time for the wordcount workload. In neither cases, the default configuration (i.e., D = 4.5 seconds, 1.5 times the heartbeat interval) leads to the best performance. This group of experiments demonstrates that for different workloads, the best delay parameter varies, indicating the necessity of parameter tuning for the delay algorithm. However, our matchmaking algorithm does not require this intricate parameter tuning process. For both workloads, the FIFO scheduler with our matchmaking algorithm achieves the lowest average response time, better than that achieved by the optimally configured delay algorithm.

Let tavg represent the average response time of all map tasks. It equals to the summation of two parts. That is,

$$t_{avg} = R_l t^l_{avg} + (1 - R_l) t^{nl}_{avg} \quad (4.2)$$

where Rl denotes the data locality rate, t^l_{avg} represents the average response time of all local map tasks, and t^{nl}_{avg} the average response time of all non-local map tasks.

Because network bandwidth is a relatively scarce resource in a MapReduce cluster [1,2] and the network data transferring rate is slower than the disk access rate when MapReduce was first developed, a local map task's execution is often much faster than that of a non-local map task. Therefore, according to Equation (4.2), increasing the data locality rate Rl tends to decrease the average response time of all map tasks tavg. On the other hand, with the delay algorithm, as the maximum delay time D increases, a job and

its tasks' execution is allowed to be delayed for a longer time. As a result, although Rl increases, both t^{l}_{avg} and t^{nl}_{avg} increase as well, leading to the potential increase of tavg. This explains why map tasks' average response time does not decrease monotonically with the increase of the maximum delay time D.

So far, we have used experiments to compare three schedulers: Hadoop FIFO scheduler, Hadoop FIFO scheduler with matchmaking algorithm, and FIFO with delay algorithm. The results show that the FIFO scheduler with matchmaking algorithm achieves the highest locality rate and the lowest map task response time without the parameter tuning hassle. Next, to further compare the delay algorithm and our matchmaking algorithm, we integrate the matchmaking algorithm into Hadoop fair scheduler and compare the following two schedulers: fair scheduler with delay algorithm and fair with matchmaking algorithm.

Figures 4.5 and 4.6 show the data locality rate and the map tasks' average response time for the Hadoop fair schedulers.

We can see that when integrated with the fair sharing scheduling, our matchmaking algorithm still achieves better data locality rates and near-optimal average response times. More importantly, our algorithm achieves this great performance without the necessity of parameter tuning.

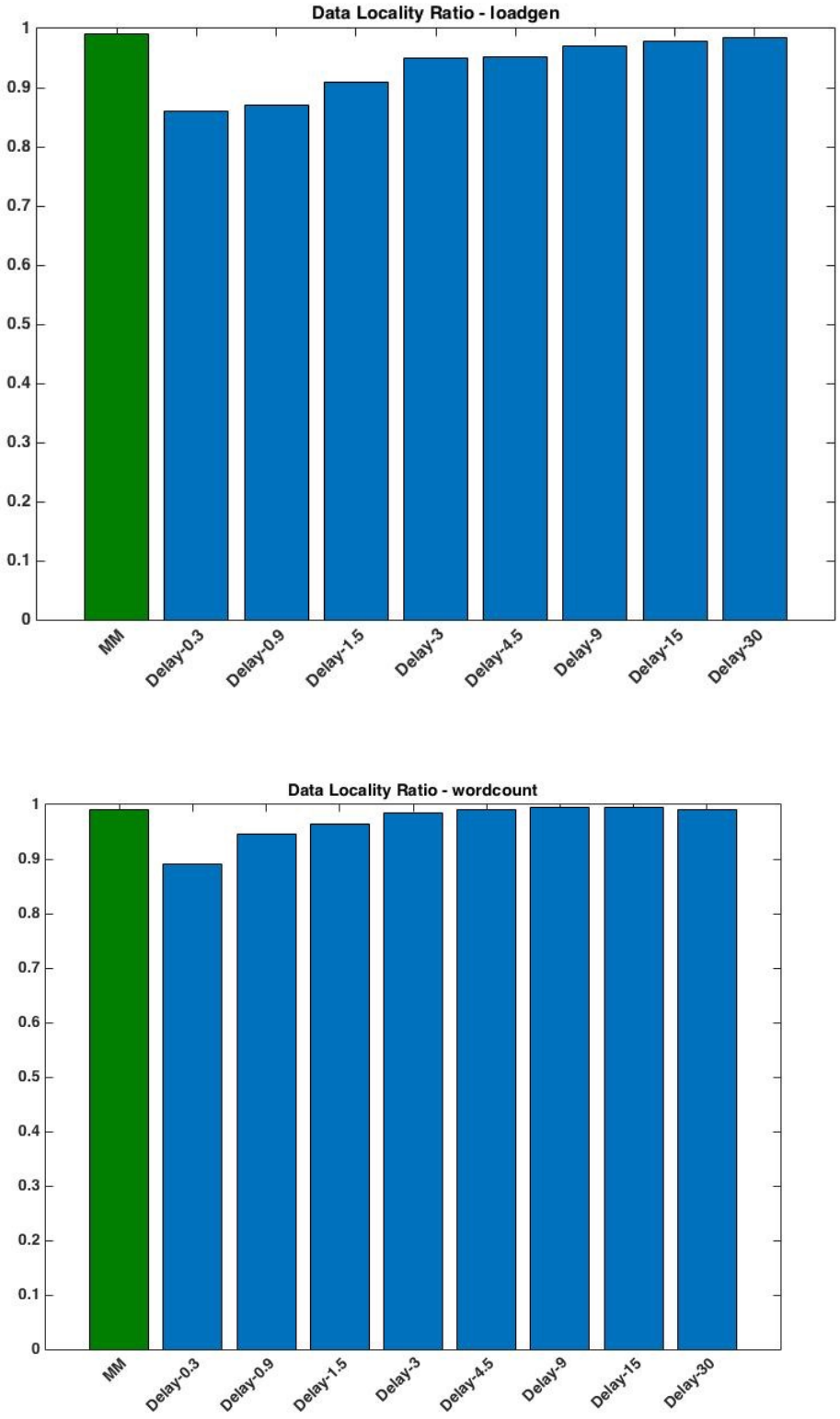
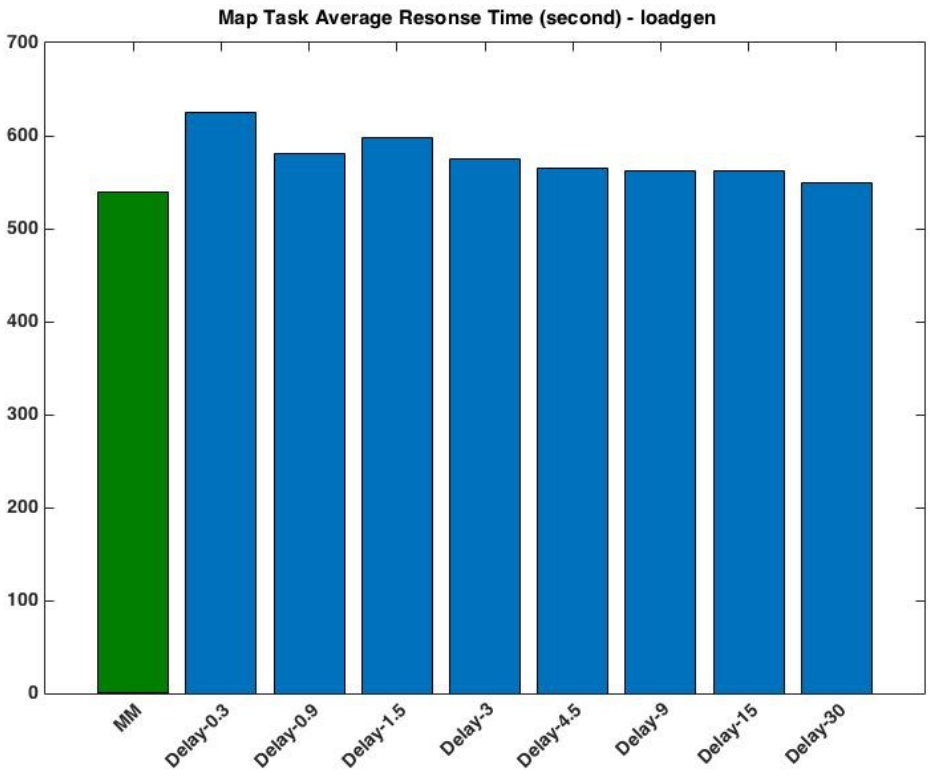


Figure 4.5 Fair Scheduler: Data Locality Rate


Map Task Average Resonse Time (second) - loadgen
700
600
500
400
300
200
100
0
MM
Delay-0.3
Delay-0.9
Delay-1.5
Delay-3
Delay-4.5
Delay-9
Delay-15
Delay-30

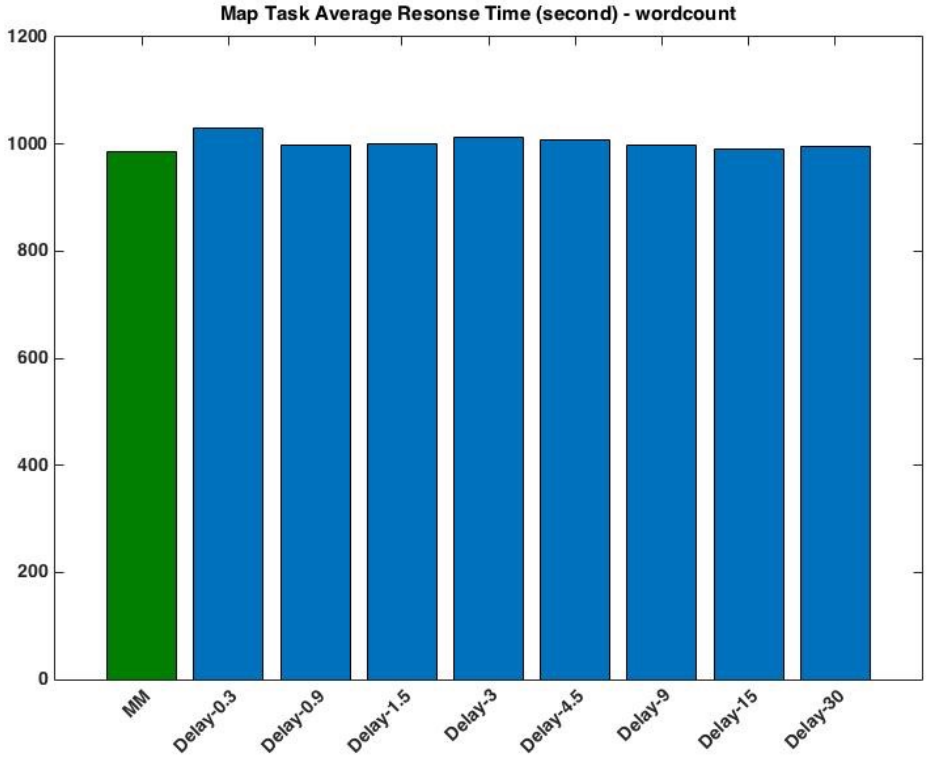


Figure 4.6 Fair Scheduler: Map Tasks' Average Response Time

CHAPTER 5. REAL-TIME MAPREDUCE SCHEDULER

With the increasing popularity of MapReduce, more and more applications were developed to employ this powerful platform. Some applications are sensitive to time. For example, financial companies require data to be processed in an acceptable time interval. A scheduler that supports real-time applications became more and more important.

In this section, we will introduce our Real-Time MapReduce (RTMR) scheduler to not only provide deadline supports for MapReduce applications executing in heterogeneous environments but also ensure good cluster utilization. The following of this section is organized as follows; first, we briefly describe the Deadline Constraint scheduler [17] and its deficiencies. And then, our scheduling algorithm is presented in detail. Evaluations of these two schedulers are provided in the end.

5.1 Deadline Constraint Scheduler

The Deadline Constraint Scheduler [17] aims to ensure deadlines for real-time MapReduce jobs. After a job is submitted, the scheduler first determines whether the job can be completed within the specified deadline or not using a schedulability test:

It assumes that all reduce tasks of a job will start executing simultaneously for the same amount of time that is known a priori. Based on this assumption, the Deadline Constraint Scheduler calculates the latest reduce start time for the job to meet its deadline. If sm is the map start time of the job, then the maximum time for the job to complete its map stage is. Unlike for the reduce stage, the Deadline Constraint Scheduler assumes that

each job executes at a minimum degree of task parallelism for the map stage. That is, the scheduler only assigns the job the minimum number of map slots that are required to meet its deadline. However, it demands all map slots to be available simultaneously at to run the job's map tasks. Assume the job's input data size is σ and the cost (i.e., time) of processing a unit data in a map task is seconds, then, the scheduler calculates as:

$$n_m^{\min} = \left\lceil \frac{c_m \sigma}{s_r^{\max} - s_m} \right\rceil . \quad (5.1)$$

The Deadline Constraint scheduler, however, has some limitations and deficiencies, which may lead to resource underutilization and deadline violations. First, because the scheduler assumes that all reduce tasks of a job start to run simultaneously, it cannot accept a job with more reduce tasks than the cluster's total number of reduce slots. Second, by checking the aforementioned two conditions in the schedulability test, the scheduler only considers a single scenario where the job's deadline might be satisfied. Those conditions are, however, unnecessary for meeting a job's deadline. Many jobs that do not pass the test can nevertheless be accepted and completed by their deadlines. For instance, even if the system does not have number of map slots available upon the job's arrival, the job can still finish its map stage on time and meet the job's deadline if we have more resources available at a later time point. Furthermore, the constraint scheduler does not consider the case where slots become available and utilized at different time points. Due to these reasons, the Deadline Constraint scheduler rejects tasks unnecessarily and cannot well utilize system resources.

Last but not least, the schedulability test conditions checked by the scheduler are insufficient to ensure the deadline constraint. As a result, accepted jobs may actually miss their deadlines, violating the hard real-time scheduler's characteristics. The cause for the deadline violation is that the scheduler only checks if a certain number of reduce slots are available at a particular time point . Instead, the job requires the specified number of reduce slots for the time interval [, D]. What could happen is that the scheduler first accepts job A because at the time when job A arrives, the system status indicates that there are reduce slots available at , and then accepts job B because we have reduce slots available at . However, the later acceptance of job B means that the job will use reduce slots for the whole time interval [, D^B] and could result in less than reduce slots being left available at and job A missing its deadline.

5.2 RTMR Scheduler

In this paper, we develop a new Real-Time MapReduce (RTMR) scheduler for heterogeneous MapReduce environments. RTMR scheduler not only provides deadline guarantees to accepted jobs but also well utilizes system resources. We have made the following three assumptions when designing RTMR scheduler:

The input data is available in Hadoop Distributed File System (HDFS) before a job starts.

No preemption is allowed. The proposed scheduler orders the job queue according to job deadlines. However, once a job starts to execute its first map task, the job will not be preempted. That is, even if a new coming job B has an earlier deadline than a currently

running job A, our scheduler makes no attempt to execute B's tasks before A's tasks.

A MapReduce job contains two stages: map and reduce stages. Similar to [21,24,26] we assume in this paper that a job's reduce stage does not start until the job's map tasks have all finished.

5.2.1 Algorithm

1) Definition

Before describing the algorithm, we first present the parameters and data structures used in RTMR scheduler.

- J=(A, D, M, R, δ): A MapReduce job J is specified by the tuple (A, D, M, R, δ), where A is the job arrival time, D is the relative deadline, M and R specify the number of map and reduce tasks for the job, respectively, and δ is the input data size of the job. For a MapReduce job, each map task processes a unique part, δ_i^m, of the job's input data, where $\sum_{i=1}^{M} \delta_i^m = \delta$.

- η: The estimated maximum ratio between a job's intermediate data size δ^r and input data size δ. That is, the input data size δ^r for the job's reduce stage is at most η*δ. For a MapReduce job, each reduce task processes a unique part, δ_i^r, of the job's intermediate data, where $\sum_{i=1}^{R} \delta_i^r = \delta^r$.

- c_m: We use c_m to denote the estimated cost (i.e., time) of processing a unit of data

in a map task. In a heterogeneous environment, $c_m^{\max}$ represents the estimated highest cost of processing a unit of data in a map task, i.e., the estimated cost that is incurred on the slowest worker node.

- c_r: We use c_r to denote the estimated cost (i.e., time) of processing a unit of data in a reduce task. Similar to $c_m^{\max}$, $c_r^{\max}$ represents the estimated highest cost of processing a unit of data in a reduce task.
- $T^m = \left[t_1^m, t_2^m, ... t_l^m \right]$: For each accepted job J, we maintain a sorted vector T^m to record the estimated available time of the cluster's map slots, after scheduling J's map tasks. In the vector, *l* denotes the total number of map slots in the MapReduce cluster.
- $T^r = \left[t_1^r, t_2^r, ... t_q^r \right]$: For each accepted job J, we maintain a sorted vector T^r to record the estimated available time of the cluster's reduce slots, after scheduling J's reduce tasks. In the vector, q denotes the total number of reduce slots in the MapReduce cluster.
- $V^m = [v_1^m, v_2^m, ... v_l^m]$: For each accepted job J, we use a sorted vector V^m to represent the actual available time of the cluster's map slots after considering J's actual execution.
- $V^r = [v_1^r, v_2^r, ... v_q^r]$: For each accepted job J, we use a sorted vector V^r to represent the actual available time of the cluster's reduce slots after considering J's actual

execution.

- ΔT: The threshold that we set for triggering the feedback controller. That is, if the difference of a job's actual and estimated finish times is larger than ΔT, RTMR scheduler will invoke the feedback controller to update waiting jobs' T^m and T^r vectors.

- ε_i^m : The execution time of the i[th] map task of job J.

- ε_i^r : The execution time of the i[th] reduce task of job J.

RTMR scheduler uses historical job execution data to estimate some of the aforementioned parameters: η, $c_m^{\max}$, and $c_r^{\max}$. After executing a job J, we could update ratio η through the following equation:

$$\eta = \max(\eta, \frac{\delta^r}{\delta}) \quad (4.2)$$

Similarly, we update the values of $c_m^{\max}$ and $c_r^{\max}$ as follows:

$$c_m^{\max} = \max(c_m^{\max}, \frac{\varepsilon_1^m}{\delta_1^m}, \frac{\varepsilon_2^m}{\delta_2^m}, \ldots \frac{\varepsilon_M^m}{\delta_M^m}) \quad (4.3)$$

$$c_r^{\max} = \max(c_r^{\max}, \frac{\varepsilon_1^r}{\delta_1^r}, \frac{\varepsilon_2^r}{\delta_2^r}, \ldots \frac{\varepsilon_R^r}{\delta_R^r}) \quad (4.4)$$

RTMR scheduler is comprised of three components. The first and most important one is the admission controller, which makes decisions on whether to accept or reject a job. The second component is the job dispatcher, which assigns tasks to execute on worker

nodes. The last component is the feedback controller. Since a job may finish at a different time than that estimated by the admission controller, when the difference is large (i.e., larger than the threshold ΔT), we use a feedback controller to update the T^m and T^r vectors of the waiting jobs. Currently, our scheduler does not consider events like node failures and re-execution of slow tasks. Consequently, deadlines might be missed in such unexpected scenarios. Therefore, we also trigger the feedback controller to keep the admission controller updated when a deadline miss happens. As a result, the admission controller could make decisions based on more accurate estimates.

2) Admission Controller

In this dissertation, we assume, for both Deadline Constraint and RTMR schedulers, that jobs are put in a priority queue following EDF (earliest deadline first) order. Our admission control mechanism is, however, applicable beyond EDF, in general, to any policy (e.g., FIFO) that defines an order in which jobs should be given resources. When a new MapReduce job arrives, the admission controller determines if it is feasible to schedule the new job without compromising the guarantees for previously admitted jobs.

Algorithms 5.1, 5.2, and 5.3 show the pseudo code of the admission control. RTMR scheduler first checks if the new job J's deadline can be satisfied or not, i.e., to check if e is not larger than A + D, where e is the estimated finish time of the job (Algorithm 5.1 lines 1-9). To estimate J's finish time, we start with identifying J's proceeding job J_p if J were inserted in the priority queue. If J were at the head of the queue, J_p is the job that

has been started latest by the dispatcher. If J is the first job submitted to the cluster, it does not have a proceeding job. Since T_p^m and T_p^r record the estimated available time of the cluster's map and reduce slots after the scheduled execution of J_p and J_p's predecessors, we can estimate job J's finish time based on these vectors. If the new job J's deadline can be satisfied, RTMR scheduler then checks whether accepting J will violate the deadline of any previously admitted job (Algorithm I lines 10-21). Since only jobs that succeed job J in the priority queue will be delayed, RTMR scheduler re-estimates their finish times. If any of them will miss deadline as a result of J's acceptance, RTMR scheduler rejects job J. Finally, once the admission controller decides to accept job J, the priority queue and the T^m and T^r vectors of J and J's successors will be updated to reflect the change (Algorithm 5.1 lines 22-23).

Table 5.1 Admission Controller

ALGORITHM 5.1. ADMISSION CONTROLLER

AC(J = (A, D, M, R, δ), Priority-Q)

// Identifying J's proceeding job J_p if J were inserted in the queue
1: J_p = getPredecessor(J, Priority-Q)
2: T_p^m = $J_p.T^m$ (T_p^m = [0,0, …0] if J_p = nil)
3: T_p^r = $J_p.T^r$ (T_p^r = [0,0, …0] if J_p = nil)
// invoke Algorithms 5.2 and 5.3 to do the calculation
4: $J.T^m = \text{Cal}\,T^m(J, T_p^m).T^m$
5: $J.T^r = \text{Cal}\,T^r(J, T_p^m, T_p^r).T^r$
6: e = $\text{Cal}\,T^r(J, T_p^m, T_p^r)$.e
7: **if** e > A + D **then**
8: **return false**
9: **end if**
10: J_p = J
11: J_s = getSuccessor(J_p, Priority-Q)
12: **while** (J_s != nil) **do**
// invoke Algorithms 5.2 and 5.3 to do the calculation
13: $T_s^m = \text{Cal}\,T^m(J_s, J_p.T^m).T^m$
14: $T_s^r = \text{Cal}\,T^r(J_s, J_p.T^m, J_p.T^r).T^r$
15: $e_s = \text{Cal}\,T^r(J_s, J_p.T^m, J_p.T^r)$.e
16: **if** $e_s > J_s.A + J_s.D$ then
17: **return false**
18: **end if**
19: J_p = J_s
20: J_s = getSuccessor(J_p, Priority-Q)
21: **end while**
22: Proiority-Q.insert(J)
23: record $J.T^m$, $J.T^r$, T_s^m and T_s^r computed above as the new T^m & T^r vectors for J and J's successors
24: **return true**

ALGORITHM 5.2. CACULATION OF T^m AND e^m

Cal T^m (J = (A, D, M, R, δ), $T^m = \left[t_1^m, t_2^m, ...t_l^m\right]$)

// This algorithm estimates e^m, job J's map stage finish time and T^m, the available time of map slots after the scheduled execution of J and J's predecessors

1: $\widetilde{\varepsilon}^m = c_m^{\max} * \max(\delta_i^m, i = 1, 2, ...M)$
2: **for k =1 to M do**
3: pick the smallest value in vector T^m **, i.e.,** t_1^m
4: t_1^m = max (t_1^m **, current Time)**
5: t_1^m += $\widetilde{\varepsilon}^m$
6: $e^m = t_1^m$
7: sort items in T^m to keep T^m a sorted vector
8: **end for**
9: **return** T^m **,** e^m

ALGORITHM 5.3. CACULATION OF T^r AND e

Cal T^r (J = (A, D, M, R, δ), $T^m = \left[t_1^m, ...t_l^m\right]$, $T^r = \left[t_1^r, ...t_q^r\right]$**)**

// This algorithm estimates e, job J's finish time and T^r, the available time of reduce slots after the scheduled execution of J and J's predecessors

// invoke Algorithm 5.2 to estimate J's map stage finish time
1: e^m = Cal T^m (J, T^m). e^m
2: $\widetilde{\varepsilon}^r = c_r^{\max} * \max(\delta_i^r, i = 1, 2, ...R)$
3: **for** k = 1 to R **do**
4: pick the smallest value in vector T^r , i.e., t_1^r
5: t_1^r = max (t_1^r , e^m)
6: t_1^r += $\widetilde{\varepsilon}^r$
7: e = t_1^r
8: sort items in T^r to keep T^r a sorted vector
9: **end for**
10: **return** T^r , e

3) Dispatcher

As mentioned in Chapter 2, a Hadoop cluster uses worker nodes to execute map and reduce tasks. Each worker node has a fixed number of map slots and reduce slots, which limit the number of map tasks and reduce tasks that a worker node can execute simultaneously. Periodically, a worker node sends a heartbeat signal to the master node. Upon receiving a heartbeat from a worker node with empty map/reduce slots, the master node invokes the scheduler to assign tasks. RTMR scheduler's dispatcher fulfills this role, allocating tasks to execute on worker nodes. Algorithm 5.4 shows the pseudo code of the dispatcher.

When jobs are inserted into the priority queue, their map stages can start and their map tasks are ready to run. Therefore, it is straightforward to dispatch map tasks following the job order/priority. No modification is needed here and RTMR scheduler dispatches map tasks following the same approach as the default Hadoop system (lines 4-5).

However, since a job's map stage finish time depends on not only the job's map stage start time but also the number of map tasks the job has, when there are multiple jobs concurrently running in the cluster, which jobs can finish their map stages and start their reduce stages earlier is not determined by the job priority alone. Although jobs start their map stages following the job order/priority, it is highly likely that jobs will not finish their map stages in that order. As a result, the reduce tasks of a lower-priority job could become ready earlier than those of a higher-priority job. Thus, if ready reduce tasks are

assigned to execute on worker nodes without any constraint, the proper execution of higher-priority jobs may be interfered by the execution of lower-priority jobs, leading to deadline violations. One simple method to avoid such interferences is to strictly enforce that jobs start their reduce stages following the job order. That is, a job cannot start the reduce stage until all proceeding jobs have finished their map stages. However, this straightforward method puts a strong constraint on job parallelism and causes inefficient utilization of system resources. Therefore, we instead design a reservation-based dispatcher, which simply ensures that a lower-priority job does not occupy slots that belong to higher-priority jobs. That is, the dispatcher reserves slots that are needed by higher-priority jobs to avoid potential interferences. Upon receiving a heartbeat from a worker node with empty reduce slots, the dispatcher assigns a reduce task to the worker node only if enough reduce slots have been left unused for higher-priority jobs (lines 6-21).

We have proved that all jobs accepted by the admission controller can be successfully dispatched and completed by their deadlines in normal scenarios when there is neither a node failure nor a task re-execution (please refer to the section 5.2.2 for the proof).

Table 5.2 Dispatcher Algorithm

ALGORITHM 5.4. DISPATCHER

DP(J=(A, D, M, R, δ), Priority-Q,i,Ra)

```
1:  m: available map slots on node i
2:  r: available reduce slots on node i
3:  Ra: the number of available reduce slots in the cluster, which is counted upon
    calling this algorithm
    // dispatch map tasks:
4:  if (m>0) then
5:      follow the same approach as the default Hadoop system to dispatch map
        tasks
    // dispatch reduce tasks:
6:  if  r > 0 then
7:     reservedSlot: the number of reduce slots reserved for high-priority jobs
8:     reservedSlot = 0
9:    for J from Priority-Q do
10:       if reservedSlot > Ra then
11:           break for
12:       end if
13:       T = findAReadyReduceTask(J)
14:       if  T != nil then
15:           assign T to node i
16:           break for
17:       else if J has not reached its reduce stage then
18:           reservedSlot += J.R
19:       end if
20:   end for
21: end if
```

4) Feedback Controller

A feedback controller is developed to keep the admission controller's records up-to-date. As described in the previous section, the admission controller makes decisions based on the job records, i.e., job's T^m and T^r vectors. These vectors record the estimated available time of the cluster's map and reduce slots after scheduling a job's execution. However, a job's actual execution may be different from the estimate. For instance, because we use $c_m^{\max}$ and $c_r^{\max}$ as the estimated cost of processing a unit of data in a map and a reduce task and η as the estimated ratio between a job's intermediate data size and input

data size, it is highly likely that some job finishes earlier than that estimated by the admission controller. In addition, node failures or speculative re-execution of slow tasks can result in a job finish time later than expected. To reduce false negatives (i.e., rejecting jobs that can meet their deadlines) and deal with unexpected events (such as node failures), a feedback controller is invoked to update all waiting jobs' T^m and T^r vectors if the difference between a job's actual and estimated finish times is larger than a certain threshold ΔT. The feedback controller is also triggered if a job misses its deadline due to unexpected events. As a result of the update, the admission controller makes decisions based on more accurate estimates. In this paper, we set the threshold ΔT to be a typical map task execution time after profiling.

ALGORITHM 5.5. FEEDBACK CONTROLLER
FC(J=(A, D, M, R, δ), Priority-Q)

1: ⊗: threshold to trigger the update
2: $\widetilde{e}$: job J's actual finish time
3: J_p = getPredecessor(J, Priority-Q)
4: T_p^m = Jp.T^m (T_p^m = [0,0, …0] if Jp = nil)
5: T_p^r = Jp.T^r (T_p^r = [0,0, …0] if Jp = nil)
// invoke Algorithm 5.3 to do the calculation
6: e = CalT^r (J, T_p^m, T_p^r).e
7: **if** | e-$\widetilde{e}$ | ≥ ⊗ or $\widetilde{e}$ > (A+D) **then**
8: build $\widetilde{E}^m$, the sorted vector containing the actual finish time of job J's map tasks
9: build $\widetilde{E}^r$, the sorted vector containing the actual finish time of job J's reduce tasks
// invoke Algorithm 5.4 to calculate the updated estimates
10: $J.T^m$ = SATU(J, T_p^m, T_p^r, $\widetilde{E}^m$, $\widetilde{E}^r$).U^m
11: $J.T^r$ = SATU(J, T_p^m, T_p^r, $\widetilde{E}^m$, $\widetilde{E}^r$).U^r
12: J_p = J
13: J_s = getSuccessor(J_p, Priority-Q)
14: **while** J_s != nil do
// invoke Algorithms 5.2 and 5.3 to do the calculation
15: $J_s.T^m$ = CalT^m (J_s, $J_p.T^m$).T^m
16: $J_s.T^r$ = CalT^r (J_s, $J_p.T^m$, $J_p.T^r$).T^r 17: $J_p = J_s$
18: J_s = getSuccessor(J_p, Priority-Q)
19: **end while**
20: **else return**
21: **end if**

ALGORITHM 5.6. SLOT AVAILABLE TIME UPDATE

SATU (J=(A, D, M, R, δ), T_p^m, T_p^r, $\widetilde{E}^m$, $\widetilde{E}^r$)

1: T_p^m : map slot available time in J's predecessor's record
2: T_p^r : reduce slot available time in J's predecessor's record
3: $\widetilde{E}^m$: sorted vector containing the actual finish time of job J's map tasks
4: $\widetilde{E}^r$: sorted vector containing the actual finish time of job J's reduce tasks
5: $U^m = T_p^m$
6: $U^r = T_p^r$
7: **while** $\widetilde{E}^m$ is not empty **do**
8: remove the item currently located at the beginning of vector $\widetilde{E}^m$, say it is $\widetilde{e}_i^m$
9: $u_1^m = \widetilde{e}_i^m$ (where u_1^m is the first and smallest item in vector U^m)
10: sort items in U^m to keep U^m a sorted vector
11: **end while**
12: **while** E^r is not empty do
13: remove the item currently located at the beginning of vector $\widetilde{E}^r$, say it is $\widetilde{e}_i^r$
14: $u_1^r = \widetilde{e}_i^r$ (where u_1^r is the first and smallest item in vector U^r)
15: sort items in U^r to keep U^r a sorted vector
21: **end while**
22: **return** U^m, U^r

Table 5.3 Feedback Controller Algorithm

5.2.2 Proof of Correctness

First, the correctness of admission control and dispatch algorithms is proved. That is, we prove that all jobs accepted by the admission controller can be successfully dispatched and completed by their deadlines in normal scenarios when there is neither a node failure nor a task re-execution. Several vector operators used in the proof are defined below.

Definition-1: > & ≥

For two sorted vectors V^A and V^B, where

$$V^A = (v_1^A, v_2^A, v_3^A, ...v_n^A), v_i^A \le v_j^A, 1 \le i < j \le n$$

and $V^B = (v_1^B, v_2^B, v_3^B, ...v_n^B), v_k^B \le v_l^B, 1 \le k < l \le n$

$V^A > V^B$ *if and only if* $v_i^A > v_i^B, i = 1,2,...n$;

$V^A \ge V^B$ *if and only if* $v_i^A \ge v_i^B, i = 1,2,...n$.

Definition-2: $\oplus$

For a sorted vector V^A

$V^A = (v_1^A, v_2^A, v_3^A, ...v_n^A), v_i^A \le v_j^A, 1 \le i < j \le n$

and a vector V^B

$V^B = (v_1^B, v_2^B, v_3^B, ...v_m^B)$

$V^A \oplus V^B$ *generates an n dimensional vector* V^C *as follows: first, let* $V^C = V^A$ *; second, from* V^B *, remove the item currently located at the beginning of the vector, say it is* v_i^B *; third, change* v_1^C *to be equal to* $v_1^C + v_i^B$ *and resort* V^C *to keep it a sorted vector; forth, repeat the second and third steps until there is no element left in* V^B.

Definition-3 maximum of a vector and a value

For a sorted vector V^A

$V^A = (v_1^A, v_2^A, v_3^A, ...v_n^A), v_i^A \le v_j^A, 1 \le i < j \le n$

and a value a, $MAX(V^A, a)$ generates an n dimensional vector as follows:

$$MAX(V^A, a) = (\max(v_1^A, a), \max(v_2^A, a), ..., \max(v_n^A, a))$$

It can be easily proved that the following properties hold for the aforementioned operators:

1) If $V^A \geq V^B$ and $V^B \geq V^C$, then $V^A \geq V^C$

2) If $V^A \geq V^B$ and $V^C \geq V^D$,

then $V^A \oplus V^C \geq V^B \oplus V^D$

3) If $V^A \geq V^B$ and $a \geq b$,

then $MAX(V^A, a) \geq MAX(V^B, b)$

The admission controller generates $J.T^m$ and $J.T^r$ vectors, which record the estimated slot available time after the scheduled execution of job J and J's predecessors, while $J.V^m$ and $J.V^r$ respectively represent the actual available time of the cluster's map and reduce slots after considering these jobs' actual execution. To guarantee that an accepted job J_i does not miss its deadline in normal scenarios, we prove $\forall i, J_i.T^m \geq J_i.V^m$ and $J_i.T^r \geq J_i.V^r$ when there is neither a node failure nor a task re-execution.

Proof-1:

Admission control algorithm ensures $\forall i, J_i.T^m \geq J_i.V^m$

For the first job J_1 admitted to the cluster, since it does not have a proceeding job,

when the admission controller calculates $J_1.T^m$, we have $T_p^m = [0,0,...0]$ (see Algorithm 1), which equals V_0^m, the initial available time of the cluster's map slots. According to Algorithm 2, $J_1.T^m$ is calculated as follows:

$J_1.T^m = MAX(T_p^m, J_1.A) \oplus \max \varepsilon_1^m = MAX(V_0^m, J_1.A) \oplus \max \varepsilon_1^m$ where $\max \varepsilon_1^m$ is a vector composed of M items with equal value of $\widetilde{\varepsilon}_1^m = c_m^{\max} * \max(J_1.\delta_i^m, i = 1,2,...M)$. In addition, we have:

$$J_1.V^m = MAX(V_0^m, J_1.A) \oplus \Sigma_1^m$$

where Σ_1^m is the vector composed of the actual execution time of J_1's map tasks. Since $\widetilde{\varepsilon}_1^m$ is a pessimistic estimation of a map task's execution time, we have:

$$\max \varepsilon_1^m \geq \Sigma_1^m$$

According to the property of vector operator "⊕", we conclude from the above three equations and inequality that:

$$J_1.T^m \geq J_1.V^m$$

Assuming $J_k.T^m \geq J_k.V^m$, we can show that $J_{k+1}.T^m \geq J_{k+1}.V^m$ following a similar proof procedure. According to mathematical induction, we conclude $J_i.T^m \geq J_i.V^m$ for all accepted job J_i.

Proof-2:

Admission control algorithm ensures $\forall i, J_i.T^r \geq J_i.V^r$

For the first job J_1 admitted to the cluster, since it does not have a proceeding job, when the admission controller calculates $J_1.T^r$, we have $T_p^r = [0,0,...0]$ (see Algorithm 1), which equals V_0^r, the initial available time of the cluster's reduce slots. According to Algorithm 3, $J_1.T^r$ is calculated as follows:

$J_1.T^r = MAX(T_p^r, J_1.e^m) \oplus \max \varepsilon_1^r = MAX(V_0^r, J_1.e^m) \oplus \max \varepsilon_1^r$ where $J_1.e^m$ is the estimated finish time of J_1's map stage and $\max \varepsilon_1^r$ is a vector composed of R items with equal value of $\widetilde{\varepsilon}_1^r = c_r^{\max} * \max(J_1.\delta_i^r, i = 1,2,...R)$. In addition, we have:

$$J_1.V^r = MAX(V_0^r, J_1.\widetilde{e}^m) \oplus \Sigma_1^r$$

where $J_1.\widetilde{e}^m$ is the actual finish time of J_1's map stage and Σ_1^r is the vector composed of the actual execution time of J_1's reduce tasks. Since as shown in ***Proof-1*** $J_1.T^m \geq J_1.V^m$, it implies the following relation for the largest items (i.e., $J_1.e^m$ and $J_1.\widetilde{e}^m$) of the two vectors:

$$J_1.e^m \geq J_1.\widetilde{e}^m$$

And since $\widetilde{\varepsilon}_1^r$ is a pessimistic estimation of a reduce task's execution time, we have:

$$\max \varepsilon_1^r \geq \Sigma_1^r$$

According to the properties of "MAX" and "⊕" operators, we conclude from the above four equations and inequalities that:

$J_1.T^r \geq J_1.V^r$

Assuming $J_k.T^r \geq J_k.V^r$, we can show that $J_{k+1}.T^r \geq J_{k+1}.V^r$ following a similar proof procedure. According to mathematical induction, we conclude $J_i.T^r \geq J_i.V^r$ for all accepted job J_i.

In the following part of this section, we prove the correctness of the feedback controller by showing that $U^m \geq J.V^m$ and $U^r \geq J.V^r$. Therefore, after updating job J's vectors T^m and T^r with U^m and U^r in Algorithm 5 (lines 10-11), the condition $J.T^m \geq J.V^m$ and $J.T^r \geq J.V^r$ (i.e., the estimated slot available time is greater or equal to the actual available time) still holds for job J.

Proof-3: **Algorithm 5.6 ensures** $U^m \geq J.V^m$

We first prove by induction that $U^m \geq V^{m,i}$ holds after the ith iteration (where i=1, …, M) of the first while loop (i.e., lines 7-11) of Algorithm 5.6. Here, $V^{m,i}$ represents how $J.V^m$ looks like after considering the actual execution of the ith map task of job J.

Step 1: $U^m \geq V^{m,i}$ is true after the first iteration of the while loop, i.e. $U^m \geq V^{m,i}$ is true for i=1.

As we have shown in ***Proof-1***, the admission control algorithm ensures $J.T^m \geq J.V^m$, therefore, after executing line 5 of Algorithm 6 (i.e., $U^m = T_p^m$) we have $U^m \geq J_p.V^m$ and thus $U^m \geq V^{m,i}$ holds before entering the while loop (i.e., $U^m \geq V^{m,i}$ is true for i=0).

Upon the completion of the first map task of job J at time point $\widetilde{e}_1^m$, $V^{m,i} = [v_1^m, v_2^m, ...v_l^m]$, the sorted vector representing the actual available time of the cluster's map slots, first gets updated to be $[v_1^m, ..., v_{j-1}^m, \widetilde{e}_1^m, v_{j+1}^m, ...v_l^m]$. Here, it is assumed that the map slot corresponding to the current j[th] position of vector $V^{m,i}$ has been used to execute the task and thus gets updated to $\widetilde{e}_1^m$. Since it takes some time to execute a task, we have the new available time greater than the old available time of the slot, i.e., $\widetilde{e}_1^m > v_j^m$. We thus know that $v_1^m \le v_2^m ... \le v_{j-1}^m \le v_j^m < \widetilde{e}_1^m$ holds, which means that for the first j items of vector $V^{m,i} = [v_1^m, ..., v_{j-1}^m, \widetilde{e}_1^m, v_{j+1}^m, ...v_l^m]$, we have $v_1^m \le v_2^m ... \le v_{j-1}^m < \widetilde{e}_1^m$. Then, we sort the vector and get $V^{m,i} = [\widetilde{v}_1^m, ..., \widetilde{v}_n^m, \widetilde{e}_1^m, \widetilde{v}_{n+1}^m, ...\widetilde{v}_{l-1}^m]$, where $n \ge j-1$. In addition, we know for $1 \le p \le j-1$, $\widetilde{v}_p^m = v_p^m$ and for $j-1 < p \le n$ and $n+1 \le p \le l-1$, $\widetilde{v}_p^m = v_{p+1}^m$.

After the first iteration of the while loop, $U^m = [u_1^m, u_2^m, ...u_l^m]$ changes to be a new sorted vector $U^m = [u_2^m, ..., u_k^m, \widetilde{e}_1^m, u_{k+1}^m, ..., u_l^m]$.

Before entering the while loop, $U^m = [u_1^m, u_2^m, ...u_l^m]$, $V^{m,i} = [v_1^m, v_2^m, ...v_l^m]$, and $U^m \ge V^{m,i}$ holds. Thus, we have for $1 \le p \le l$, $u_p^m \ge v_p^m$. After the aforementioned updates, we have $U^m = [u_2^m, ..., u_k^m, \widetilde{e}_1^m, u_{k+1}^m, ..., u_l^m]$ and $V^{m,i} = [\widetilde{v}_1^m, ..., \widetilde{v}_n^m, \widetilde{e}_1^m, \widetilde{v}_{n+1}^m, ...\widetilde{v}_{l-1}^m]$, and

For the first k-1 items of the two vectors**,** i.e., when $1 \le p \le k-1$, $u_{p+1}^m \ge \widetilde{v}_p^m$ holds. The reasoning is as follows: $\widetilde{v}_p^m$ equals either v_p^m or v_{p+1}^m. When $\widetilde{v}_p^m = v_p^m$, because $u_{p+1}^m \ge u_p^m$,

$u_p^m \geq v_p^m$, and $v_p^m = \tilde{v}_p^m$, we have $u_{p+1}^m \geq \tilde{v}_p^m$; and when $\tilde{v}_p^m = v_{p+1}^m$, because $u_{p+1}^m \geq v_{p+1}^m$ and $v_{p+1}^m = \tilde{v}_p^m$, we too have $u_{p+1}^m \geq \tilde{v}_p^m$.

The k^th^ item of U^m is always greater or equal to that of $V^{m,i}$. The reasoning is as follows: because when $1 \leq p \leq k-1$, $u_{p+1}^m \geq \tilde{v}_p^m$ and both $V^{m,i} = [\tilde{v}_1^m,...,\tilde{v}_n^m,\tilde{e}_1^m,\tilde{v}_{n+1}^m,...\tilde{v}_{l-1}^m]$ and $U^m = [u_2^m,...,u_k^m,\tilde{e}_1^m,u_{k+1}^m,...,u_l^m]$ are sorted vectors, $\tilde{e}_1^m$'s position in U^m must be earlier than that in $V^{m,i}$, i.e., $k \leq n+1$. If $k = n+1$, the k^th^ items of vectors U^m and $V^{m,i}$ all equal to $\tilde{e}_1^m$. If $k < n+1$, the k^th^ items of vectors U^m and $V^{m,i}$ are $\tilde{e}_1^m$ and $\tilde{v}_k^m$. Since $V^{m,i} = [\tilde{v}_1^m,...,\tilde{v}_n^m,\tilde{e}_1^m,\tilde{v}_{n+1}^m,...\tilde{v}_{l-1}^m]$ is a sorted vector, i.e., $\tilde{v}_1^m \leq ... \leq \tilde{v}_k^m \leq ... \leq \tilde{v}_n^m \leq \tilde{e}_1^m$, we have $\tilde{e}_1^m \geq \tilde{v}_k^m$. That is, the k^th^ item of U^m is always greater or equal to that of $V^{m,i}$.

For all items from the (k+1)^th^ to the n^th^ positions, i.e., when $k+1 \leq p \leq n$, we have $u_{p+1}^m \geq \tilde{v}_p^m$ since $u_{p+1}^m \geq \tilde{e}_1^m$ and $\tilde{e}_1^m \geq \tilde{v}_p^m$.

The (n+1)^th^ item of U^m is always greater or equal to that of $V^{m,i}$. The reasoning is as follows: we know that $k \leq n+1$. If $k = n+1$, the k^th^ items of vectors U^m and $V^{m,i}$ are equal since they both equal to $\tilde{e}_1^m$. If $k < n+1$, the (n+1)^th^ items of U^m and $V^{m,i}$, are u_{n+1}^m and $\tilde{e}_1^m$ respectively. Since $U^m = [u_2^m,...,u_k^m,\tilde{e}_1^m,u_{k+1}^m,...,u_l^m]$ is a sorted vector, i.e., $u_2^m \leq ... \leq u_k^m \leq \tilde{e}_1^m \leq ... \leq u_{n+1}^m \leq ...$

$\leq u_l^m$, we have $u_{n+1}^m \geq \tilde{e}_1^m$, the (n+1)^th^ item of U^m is greater or equal to that of $V^{m,i}$.

For the last l-(n+1) items of the two vectors, i.e., when n+1 ≤ p ≤ l-1, we have $u_{p+1}^m \geq \widetilde{v}_p^m$ since $u_{p+1}^m \geq v_{p+1}^m$ and $\widetilde{v}_p^m = v_{p+1}^m$.

In summary, $U^m \geq V^{m,i}$ holds after the first iteration of the while loop, i.e., $U^m \geq V^{m,i}$ is true for i=1.

Step 2: Assume $U^m \geq V^{m,i}$ holds after the q[th] iteration of the while loop, i.e., $U^m \geq V^{m,i}$ is true for i=q.

Step 3: Following a procedure similar to Step 1, we can prove that $U^m \geq V^{m,i}$ also holds after the (q+1)[th] iteration of the while loop, i.e., $U^m \geq V^{m,i}$ is true for i=q+1.

According to mathematical induction, we conclude $U^m \geq V^{m,i}$ holds after the i[th] iteration, for i=1, …, M, of the first while loop (i.e., lines 7-11) of Algorithm 6.

Since the values of both vectors (i.e., U^m and $V^{m,i}$) do not change after the first while loop, we have proved that Algorithm 6 ensures $U^m \geq V^{m,i}$ for i=M, that is, $U^m \geq J.V^m$.

Proof-4: Algorithm 5.6 ensures $U^r \geq J.V^r$

Similar to the procedure of Proof-3, we can prove Algorithm 5.6 ensures $U^r \geq J.V^r$.

According to Proof-3 and Proof-4, we conclude that after updating $J.T^m$ and $J.T^r$ with U^m and U^r by invoking Algorithm 5.6 in Algorithm 5.5, the condition $J.T^m \geq J.V^m$ and $J.T^r \geq J.V^r$ (i.e., the estimated slot available time is greater or equal to the actual available time) still holds for job J.

5.3 Evaluation of RTMR scheduler and Deadline Constraint Scheduler

Our implementation of Deadline Constraint scheduler and RTMR scheduler are all based on the Hadoop 0.21. These two schedulers are implemented and compared experimentally in terms of real-time property and cluster utilization. To test the effects of feedback control, we run RTMR scheduler twice, with and without the feedback controller enabled. In addition, since the cluster utilization is determined by not only the scheduling algorithm but also the workload volume, we run the default Hadoop FIFO scheduler, which accepts all jobs to execute in the cluster, collecting its resultant cluster utilization to reflect the workload volume. If a real-time scheduler achieves a cluster utilization close to that achieved by the default Hadoop FIFO scheduler, we **consider** that the resource cost of providing the real-time property is not high.

For the RTMR scheduler, the admission controller is implemented in the *JobQueueJobInProgressListener* class, which makes the admission control decision and maintains the MapReduce job queue. The dispatcher is in the *RTMRTaskScheduler* class, which extends from the *TaskScheduler* class and is in charge of dispatching map and reduce tasks. The feedback controller is also in the *JobQueueJobInProgressListener* class, where we set the threshold Δ to be a typical map task execution time.

Similarly, Deadline Constraint scheduler's admission controller is in *JobQueueJobInProgressListener* class and its dispatcher, called *DCTaskScheduler*, extends from the *TaskScheduler* class.

5.3.1 Experimental Environment

We have evaluated RTMR scheduler and compared it with Deadline Constraint Scheduler [21] in a heterogeneous Hadoop cluster that contains one master node and 30 worker nodes.

Table 5.4 Experimental Environment

Nodes	Quantity	Hardware and Hadoop Configuration
Master node	1	2 single-core 2.2GHz Opteron-248 CPUs, 8GB RAM, 1Gbps Ethernet
Type 1 worker nodes	20	2 dual-core 2.2GHz Opteron-275 CPUs, 4GB RAM, 1 Gbps Ethernet, 4 map and 1 reduce slots per node
Type II worker nodes	10	2 single-core 2.2GHz Opteron-64 CPUs, 4GB RAM, 1 Gbps Ethernet, , 2 map and 1 reduce slots per node

A heterogeneous Hadoop cluster that contains one master node and 30 worker nodes is used as the testbed. The 30 worker nodes are configured as one rack and they are of two types. 20 of them are 2 dual-core CPU nodes and 10 of them are 2 single-core CPU nodes. Table I gives the detailed hardware information of the cluster. We make the number of map slots in a worker node equal to the number of CPU cores. Because each node has only one Ethernet card, we configure one reduce slot per worker node to avoid bandwidth competition between multiple reduce tasks on a single node. *Loadgen,* a test example in Hadoop source code for evaluating Hadoop schedulers [40], is used as the test application.

5.3.2 Workload and Experiments

We first create a submission schedule (workload I) that is similar to the one used by

Zaharia et al [40] that was described in Chapter 4.

Table 5.5 Workload I

Bin	#Maps	%Jobs at Facebook	#Maps in Benchmark	# of jobs in Benchmark
1	1	39%	1	38
2	2	16%	2	16
3	3-20	14%	10	14
4	21-60	9%	50	8
5	61-150	6%	100	6
6	151-300	6%	200	6
7	301-500	4%	400	4
8	501-1500	4%	800	4
9	>1501	3%	4800	4

Table 5.6 Workload I's Configuration (in Terms of Number of Map, Reduce Tasks and Deadline)

Bin	#Maps	#Reduces	Deadline (second)
1	1	[1,5]	[200,300]
2	2	[1,5]	[200,300]
3	10	[5,10]	[300,400]
4	50	[10,20]	[500,800]
5	100	[20,30]	[1000,1500]
6	200	30	[2000,2500]

Since most jobs in the Facebook workload are small, in particular, some of them having only 1 map task, we create workload II to include more jobs with higher parallelism. That is, in workload II, we let the number of map tasks per job follow a normal distribu-

tion with an average of 100. Again, because of the moderate size of our cluster, we do not include the three jobs that have more than 300 map tasks. Table 5.7 shows the detailed information of workload II. To test how RTMR scheduler works with large jobs, we also create some jobs with more reduce tasks than the cluster's total number of reduce slots in workload II. However, since we already know that Deadline Constraint scheduler cannot accept such jobs, they are not included in workload II when Deadline Constraint scheduler is tested.

For performance evaluation of the real-time schedulers, the following three metrics, i.e. *job accept ratio, job success ratio*, and *cluster utilization* are used:

$$AcceptR = \frac{\#accepted\_jobs}{\#jobs\_in\_a\_workload}$$

$$SuccessR = \frac{\#successful\_jobs}{\#accepted\_jobs}$$

$$Util = \frac{slot\_time\_used\_by\_successful\_jobs}{available\_slot\_time\_during\_workload\_exe}$$

Table 5.7 Workload II

Bin	No. Job	#Maps	#Reduces	Deadline (second)
1	9	[1,10]	[1,5]	[200,300]
2	24	[10,50]	[5,10]	[300,500]
3	25	[50,100]	[15,30]	[1000,1500]
4	18	[100,200]	[25,50]	[1500,2500]
5	13	[200,300]	[35,70]	[2500,3500]

The following equation is used to calculate the cluster utilization achieved by default

Hadoop FIFO scheduler:

$$Util = \frac{slot\_time\_used\_by\_all\_jobs}{available\_slot\_time\_during\_workload\_exe}$$

Here, *successful_jobs* denote those jobs that finish before their deadlines and *slot_time_used_by_successful_jobs* refer to the total map and reduce slot time used to execute them. Since Hadoop FIFO scheduler does not consider job deadlines and provides no real-time guarantees, it accepts all jobs and its cluster utilization is calculated using *slot_time_used_by_all_jobs* instead.

available_slot_time_during_workload_exe refers to the total usable time of cluster map and reduce slots during the execution of a workload, i.e., the product of the number of slots and the turnaround execution time of all accepted jobs in a workload.

Tables 5.8 and 5.9 show how the tested schedulers perform with workload I and II respectively. As we can see, although compared to RTMR scheduler Deadline Constraint scheduler accepts more jobs, it fails to provide deadline guarantees to all accepted jobs, with job success ratio of 85.7% and 22.5% respectively. Since not all accepted jobs are successful while more jobs are accepted, which prolong the workload's execution in the cluster, Deadline Constraint scheduler leads to much lower cluster utilizations of only 5.7% and 0.7% respectively. In contrast, RTMR scheduler maintains good cluster utilization of 15.5% and 64.6%, in comparison to 21.3% and 69.7% achieved by default Hadoop FIFO scheduler. Deadline Constraint scheduler's very poor performance with workload II experimentally demonstrates its deficiencies in handling real-time MapRe-

duce jobs with high parallelism. From the data, we can also conclude that RTMR scheduler performs better when we enable the feedback controller to keep the admission controller up-to-date, which results in better *job accept ratio* and *cluster utilization*.

Table 5.8 Scheduler Performance with workload I

Metrics	Constraint Scheduler	RTMR Scheduler	RTMR No Feedback	FIFO Scheduler
Accept Ratio	71.6%	56.8%	46.6%	n/a
Success Ratio	85.7%	100%	100%	n/a
Cluster Utilization	5.7%	15.5%	11.6%	21.3%

Table 5.9 Scheduler Performance with Workload II

Metrics	Constraint Scheduler	RTMR Scheduler	RTMR No Feedback	FIFO Scheduler
Accept Ratio	44.9%	24.7%	15.7%	n/a
Success Ratio	22.5%	100%	100%	n/a
Cluster Utilization	0.7%	64.6%	49.8%	69.7%

The FIFO scheduler has highest utilization in both Workload I and Workload II. This is because FIFO scheduler does not reject any job. Our RTMR scheduler achieve second highest utilization. It is because the Feedback controller helps RTMR to accept more jobs that can be finished before their deadlines.

Through the development of RTMR scheduler, we well understood how to support applications/jobs that have SLA requirements in Hadoop MapReduce clusters. It helps us

deep understand Hadoop MapReduce features, mechanisms, and patterns. For our next step research, an energy efficient scheduler will be developed based on the knowledge obtained from these achievements.

CHAPTER 6. ENERGY EFFICIENT SCHEDULER

With the increasing demands of computational power in big data analytics, Hadoop cluster becomes larger and larger and the maintenance cost rises correspondingly. How to improve a Hadoop cluster's computational power with sustainable costs is a big challenge. To resolve this problem, scientists introduced GPU into Hadoop cluster [121,123,124,129,130,131]. However, scheduling MapReduce applications in hybrid CPU-GPU clusters has not been systematically studied. The remaining work will focus on this problem.

In this paper, we will build an energy-efficient scheduler in a hybrid MapReduce environment. we must consider several factors simultaneously. After analyzing this problem carefully, we propose our energy consumption model and list the challenges that need to be resolved in building a two-level energy-efficient scheduler.

6.1 Background

Since scheduling MapReduce applications in a hybrid Hadoop cluster is complicated, in this section, we first present the background information to explain the challenges.

6.1.1 YARN label scheduling

In Chapter 2, we have described how YARN framework works. With the increasing demand for computational power, different types of hardware can be added to it. It means NodeManger needs to run on various types of servers. To provide a flexible scheduling

mechanism, the Hadoop community introduced label scheduling.

In a label scheduling algorithm, Hadoop administrators can give different types of labels to different NodeManagers. Customers/users should be able to know these labels before submitting applications and thus submit applications with the proper labels. For example, there are 3 NodeManagers: A, B, and C. A has label-1 and label-2, B has label-2, and C has no label. There is a user submitting an application with label-2. Then, this application can only run on NodeManager A and B. That is, resources on NodeManagers that do not have label-2 are not allowed to run this application. But an application without a label can run on any NodeManager. NodeManagers that have no label can run applications that have no label requirement or require no label NodeManagers.

With the help of a label scheduling mechanism, we give label-1 to a CPU node and label-2 to a GPU node. All applications that require GPU (aka. GPU application) will be given label-2 during submission. It means a GPU application can only run on a NodeManager that has GPU in place. For CPU applications, they are able to run on CPU nodes as well as GPU nodes (where they will only use CPUs of GPU nodes and leave GPUs idle).

In next section, we present assumptions we have for our research work. And then, we provide a solution called adaptive execution to instead allow GPU applications to run on both GPU and CPU nodes.

6.1.2 Assumptions

In most of the research works in the cluster power management field, two constraints

need to be considered simultaneously. One is the energy consumption which scientists/ researchers want to optimize. The other is throughput or response time that is related to the applications turnaround time. Research work may become impractical if we only consider first constraint but neglect the second one. Take a small cluster as example, assume we have a 3-node cluster: node A, B, and C. Node A is the most energy efficient comparing with node B and node C. To minimize the energy consumption without considering throughput, we can simply turn off node B and node C because node A is the most energy efficient server which has the lowest cost to do the same computation comparing with other two nodes. In this way, we saved energy but applications take more time to finish.

However, this is not an acceptable balance between turnaround time and efficiency. Based on these common sense, we need to clarify three important assumptions that are critical to our research work.

1. All servers of our cluster are always powered up, in this dissertation we assume all servers are up all the time.

2. GPUs can only run one task at a time and there is no preemption or time-sharing inside a GPU.

3. We only consider map task scheduling and only map tasks will utilize GPU. The reason is that reduce tasks include shuffle which is I/O intensive. To run an I/O intensive task on a GPU may not be faster and is energy inefficient. Data needs to be moved from main memory to GPU memory before using GPU and moved out from GPU memory to main memory before and after running a task on a GPU.

Since we are providing a Hadoop system for applications to run, customers/users have to write their program in a particular way to utilize a GPU. That is, a GPU program is using specific libraries and instruction sets and can only run on a GPU. Current technology does not support adaptive execution to allow an application to run on both GPU and CPU. In this work, we will propose our adaptive execution mechanism to enable that.

6.1.3 Adaptive Execution

From the introduction in Chapter 1, we know that GPUs can save more energy than CPUs when run some applications. However, GPUs are more expensive and applications that are capable to utilize GPUs require more time to develop. Consequently, current Hadoop MapReduce clusters have less GPU nodes than CPU nodes. In some extreme case, if a cluster has a very limited number of GPU nodes, a GPU application may suffer starvation or experience a long waiting time. To resolve this problem, we have proposed a solution called: adaptive execution. It helps customers/users leverage both GPU and CPU to run their applications. It works as follows:

1. Customers/users need to include both CPU code and GPU code in their program.

2. Customers/users need to add a device checking module in their program so that it can detect whether a node has an idle GPU or not. If so, it will run the GPU code. Otherwise, it runs the CPU code. Without the GPU only constraint, customers/users no longer need to specify a label for their program during the job submission.

Instead of waiting for GPU nodes, adaptive execution provides a positive effect on the

job execution in a parallelized and load-balanced data processing platform like MapReduce. For example, a task of a MapReduce job, which has 3 tasks in total, takes 10 minutes running on a GPU node but 20 minutes on a CPU node. However, there is only 1 GPU node available at that moment. Then, if the scheduler only uses the GPU node, this job will take about 30 minutes to finish. However, if the scheduler assigns 2 tasks to the GPU node and one task to a CPU node, it takes 20 minutes. In this way, we saved energy and optimized turnaround time at the same time.

In Table 6.1, we demonstrate how to enable adaptive execution in a MapReduce application. We have two methods: *mapOnGPU()* and *mapOnCPU()*. When a map task is dispatched, it will automatically run the *setup()* method. In the *setup()* method, it detects whether the current node has idle GPU or not. If so, it sets the *hasGPU* flag to be "true". Then, when running the *map()* method, the "*hasGPU*" is used to decide which set of code should be executed. If "true", then, it calls *mapOnGPU()*. Otherwise, It calls *mapOnCPU()*.

The process described in Table 6.1 happens after a map task is dispatched. In our evaluation, we will test our scheduler with and without adaptive execution to demonstrate the difference.

Table 6.1 Adaptive Execution for MapReduce Application

Example: Adaptive Execution in MapReduce Application

```
//Beginning of MapReduce Application
1:  public class MapReduceApp {
2:       //mapper class
3:         public static class MapClass extends MapReduceBase implements
Mapper<Writable, Writable> {
4:          //flag to identify whether a node has GPU or not
5:        boolean hasGPU = false;
6:           //setup environment before map stage
7:           setup() { if (node has idle GPU) then hasGPU = true}
8:           // map method
9:           public void map()  {
10:                if (hasGPU) then run mapOnGPU();
11:                else run mapOnCPU();
12:                end if
13:           }
14:           //map code run on GPU
15:         void mapOnGPU() { do map on GPU using GPU code}
16:           //map code run on CPU
17:           void mapOnCPU() {do map on CPU using CPU code}
18:       }
19:          …….....
20:       //rest of the MapReduce application
21:  }
```

6.1.4 Relevant Container

In this section, we will introduce a new concept: relevant container. It is used in our scheduling algorithm as the resource unit. Since we have 2 types of jobs; CPU job and GPU job, the relevant container concept is used by the scheduler to allocate appropriate resources to different types of jobs.

For a task in a given CPU job, the relevant container means the CPU and memory resources that can be used to execute this task. Similarly, for a task in a GPU job, the relevant container means the GPU, memory, and CPU resources that are needed. For example, there is a 2-node idle cluster which has one CPU node with 4 CPU containers and one

GPU node with 1 GPU container (contains 1 GPU and 1 CPU resources) and 3 CPU containers. For a CPU job *j* which requests 3 containers, there are 8 relevant containers because it does not need GPU. For a GPU job *g* which requests 2 GPU containers, there are two scenarios:

1. Without adaptive execution, job *g* has only one relevant container.

2. With adaptive execution, job *g* has 8 relevant containers.

Our scheduling algorithm chooses relevant containers while considering energy cost, data locality, and performance.

6.2 Scheduling Algorithm

There are three aspects that need to be considered simultaneously when we design an energy-efficient MapReduce scheduler: energy minimization, data locality, and QoS control. Based on our assumptions and adaptive execution, both CPU and GPU applications can run on CPU and GPU nodes. Thus, we do not need to involve label scheduling in the case where adaptive execution is applied.

Resource scheduling is a match-making and bin-packing problem which is NP-hard. How to match MapReduce jobs to energy efficient containers becomes a critical problem for us to resolve. In this work, we introduce a heuristic function to facilitate our decision-making:

$$h_{ij} = ef_{ij} * dop_{ij} \quad (6.1)$$

$$dop_{ij} = (\text{input data on node i for job j}) / (\text{total data requested by job j}) \quad (6.2)$$

$$ef_{ij}=1/E_{ij\_map} \quad (6.3)$$

E_{ij_map} is the energy consumption for job j's map task running on container i. ef_{ij} is the energy efficiency of a given container i running job j's map task. dop_{ij} is the job's data overlap percentage on the node of the container and h_{ij} is the fitness score.

We can get E_{ij_map} for all types of map tasks running on all kinds of containers by profiling a single map task of job j on all containers. Based on our first assumption in section 6.1.2, we do not turn off any server. It means idle energy consumption is always there. Then, E_{ij_map} can be computed as shown in the following formula:

$$E_{ij\_map}= E_{ij\_total} - P_{idle}*T \quad (6.4)$$

E_{ij_total} is the total energy consumed by a single map task of job j running on container i in time T. E_{ij_total} can be obtained by running a single map task of job j on a node which only runs container i. T is the execution time of job j's single map task and P_{idle} is the idle power consumption of the node. In this work, a kill-a-watt meter [148] is used to collect the energy consumption and measure the idle power consumption.

In a MapReduce cluster, it is possible that a server that has the most input data of a job may not be energy efficient to run the job. As we have described in Chapter 2, data locality is important for reducing unnecessary data transfer within a cluster when a Hadoop application is running. For a MapReduce application, high data locality percentage means shorter average map task response time. In this work, dop_{ij} is introduced to measure this data locality feature of a container. When a job is submitted dop_{ij} will be computed and

saved in memory during the whole life cycle of an application. When allocating containers to a job, the fitness score h_{ij} is used by the scheduler to rank all candidate containers and to make the decision.

To achieve high data locality and energy efficiency, we need to find the best candidate container to run a task of a given job. However, it is impractical to let job/task wait till the best candidate container becomes available since it will lead to Hadoop cluster throughput degradation. By considering these requirements on data locality, energy efficiency, and QoS, we have designed and developed our algorithm.

In the following two sections, we will describe our work in developing a two-level energy-efficient mapreduce scheduler.

6.2.1 Level I: Application Scheduler

The RM (ResourseManager) level scheduling, which we refer to as application scheduler, is important since it determines if a MapReduce job can get the most energy-efficient containers or not. Table 6.2 shows how our application scheduler works.

When job j is submitted, the job scheduler will calculate the fitness scores for this job. Assuming we already know the energy efficiency for running job j on container i, we have the fitness score by simply multiplying dop_{ij} and ef_{ij}. The scheduler will rank all the containers in the cluster according to the fitness score h_{ij}. Then, the scheduler creates an $optset_j$ for job j. It is a collection of containers from slave nodes in the cluster. The scheduler picks containers from slave nodes in non-increasing order of fitness score h_{ij}. For

example, job *j* requests 3 containers. If node *k* has 2 containers with fitness score of 3 and node *l* has 3 containers with fitness score of 2. The $optset_j$ will have 3 containers in which two of them are from node *k* and 1 container is from node *l*. Note, a node's CPU container and GPU container could have different fitness scores for a job.

However, since we assume a shared environment and there may be other jobs running in the cluster, it is possible that the currently available containers are not in the job's *optset*. To help making the selection, we introduce a ranking factor rj :

rj= (*currently available containers in* $optset_j$) / (*total No. of containers of a job in* $optset_j$)

After getting rj, the scheduler will reorder all jobs in the job queue in non-increasing order of their ranking factors rj. This way, we will first schedule jobs with the most amount of desirable resources available and we have ordered jobs by considering both energy efficiency and data locality.

For each submitted job *j*, we try to assign containers in its $optset_j$ as many as possible since this can provide better energy consumption according to our algorithm. However, it is possible that the cluster is busy and does not have enough idle resources to meet job *j'*s resource requirement at a certain moment. Then, starvation or long job delay may happen.

To avoid job starvation, we introduce two queues. One is for newly submitted jobs and the other is for jobs that have waited longer than a given threshold. When a job is submit-

ted, it will go to the first queue. However, according to our scheduling algorithm, it is possible that a job is ranked low and has not been picked for a long time. To job avoid starvation, we introduce a delay time threshold. The time counter starts once a job is submitted and put in the job queue. If a job waits longer than a given threshold, we will move it to the long waiting queue. Every time there is a node heartbeat, the scheduler will check whether this long waiting queue has a job or not. If so, it will first schedule a job from the long waiting queue. If there is no job in the long waiting queue, the scheduler starts to consider the original queue. This way, we prioritize the jobs considering their ages to avoid starvation.

In Table 6.2, we did not specify label scheduling because we assume customers/users implement adaptive execution. With adaptive execution, both CPU and GPU applications can run on any NodeManager. The following algorithm is invoked every time when a job releases resources, i.e., more resources become available.

Table 6.3 demonstrates how the Algorithm works without adaptive execution. If we do not have adaptive execution, GPU jobs can only run on GPU containers. Then, when computing $optset_j$ and container allocation, the scheduler needs to first check $Label_j$ that is associated with job j. Only nodes that have $Label_j$ will be considered.

Table 6.2 Application Scheduler with Adaptive Execution

Algorithm 1: Application Scheduler with Adaptive Execution

Rj: the number of containers that job j requests, for CPU job, it requests CPU containers, for GPU job, it requests GPU or CPU containers
R_r: currently available CPU and GPU containers in the cluster
dop_{ij}: data overlap percentage for job *j* on container *i*
h_{ij}: fitness score for job *j* running on container *i*
r_j: ranking factor of job *j*
$optset_j$: optimal-set for job *j*
T: waiting time threshold

```
1:  for each job j in the job queue          //initialize optimal-set for all jobs
2:      for each relevant container i in the cluster
3:          calculate dop_ij and h_ij
4:      end for
5:      rank all containers in non-increasing order of h_ij
6:      optset_j= the first Rj containers
7:  end for
8: for each job j in the job queue      //calculate r_j for each job
9:    count=0
10:    for each relevant container k in R_r
11:       if (container k is in optset_j)   then
12:           count++
13:       end if
14:    end for
15:    r_j = count/Rj
16: end for
17: if job j's waiting time exceeds T then
18:        move job j to the long-waiting job queue
19: end if
20: sort both job queues (the regular job queue & the long-waiting job queue) respectively in non-increasing order of r_j
//assign resources to jobs in long-waiting queue
21: for each job j in the long-waiting job queue
22:     if (R_r .relevantContainers.size() >= R_j) then
23:          assign job j the currently available best R_j relevant containers and remove them from R_r
24:          update R_j
25:     else
26:          assign job j all the relevant containers that are currently available and remove them from R_r and update R_j
27:     end if
28: end for
29: if (the long-waiting job queue is empty) then
30:     for each job j in the regular job queue
31:         if (R_r .relevantContainers.size() >= R_j) then
32:              assign job j the currently available best R_j relevant containers and remove them from R_r
33:              update R_j
34:         else
35:             break
36:         end if
37:     end for
38: end if
```

Table 6.3 Application Scheduler without Adaptive Execution

Algorithm 1: Application Scheduler without Adaptive Execution

Rj: the number of containers that job *j* requests, for CPU job, it requests for CPU containers, for GPU job, it requests for GPU containers
R_r: currently available CPU and GPU containers in the cluster
dop_{ij}: data overlap percentage for job *j* on container *i*
h_{ij}: fitness score for job *j* running on container *i*
r_j: ranking factor of job *j*
$optset_j$: optimal-set for job *j*
$Label_j$: job *j's* label (only GPU job has a label)
T: waiting time threshold

```
1:  for each job j in the job queue  //initialize optimal-set for all jobs
2:      for the relevant containers that has Label_j in the cluster
3:          calculate dop_ij and h_ij
4:      end for
5:      rank all relevant containers have in non-increasing order of h_ij
6:      optset_j= the first Rj relevant containers
7:  end for
8: for each job j in the job queue        //calculate r_j for each job
9:    count=0
10:    for each container k in R_r that has Label_j
11:       if (container k is in optset_j)   then
12:           count++
13:       end if
14:    end for
15:    r_j = count/Rj
16: end for
17: if job j's waiting time exceeds T then
18:        move job j to the long-waiting job queue
19: end if
20: sort both job queues (the regular job queue & the long-waiting job queue) respectively in non-increasing order of r_j
//assign resources to jobs in long-waiting queue
21: for each job j in the long-waiting job queue
22:     if (R_r .relevantContainers.size() >= R_j) then
23:         assign job j the relevant containers that are available and remove them from Rr
24:         update R_j
25:    else
26:       assign job j relevant containers and remove them from Rr, update Rj
27:     end if
28: end for
29: if (the long-waiting job queue is empty) then
30:     for each job j in the regular job queue
31:         if (R_r.relevantContainers.size() >= R_j) then
32:             assign job j the best R_j available relevant containers and remove them from Rr
33:                 update R_j
34:         else
35:             break
36:         end if
37:     end for
38: end if
```

6.2.2 Level II: Task Scheduler

Once ApplicationMaster (AM) gets containers from RM, it will sort Job *j*'s tasks in a non-decreasing order of L_k,

$$L_k = \textit{the number of task k's local containers} \quad (6.5)$$

For job *j*, we sort tasks with positive L_k in a non-decreasing order of L_k because the task which has less local containers should be assigned first. For example, task A only has one local container (container A), however, task B has two local containers (container A and container B). If we accidentally assign task B to container A, task A has no local container anymore since one container runs one task at a time. To get a higher data locality ratio, we should schedule task A to container A and task B to container B. For tasks that have no locality, that is $L_k = 0$, we always put them in the end of the task queue. That is, non-local tasks will be scheduled only if there is no local task remaining in the task queue.

Another optimization which enhances the energy efficiency is to sort available containers according to the fitness scores when dispatching tasks to the containers. In Algorithm 2, we first sort containers according to their fitness score h_{ij}. Once AM dispatcher finishes sorting containers, it picks a task from the task queue and searches all available containers. It only assigns the task to a container under the following two conditions:

1. The container's node stores the data needed by a task;

2. Current task's $L_k = 0$. It means there is no local task in the task queue. We should start to assign these non-local tasks;

Table 6.4 Task Scheduler: Dispatcher

Algorithm 2: Application Master (AM) Dispatching (job j)'s Tasks

```
// This algorithm is invoked when AM obtained containers from RM and starts to dispatch tasks.
C_R: AM obtained a collection of containers from RM in current scheduling period in Algorithm 1
L_k: locality factor =  number of task k's local container(s) in C_R
Q: task queue
1:  sort task queue in non-decreasing order of L_k (except tasks with L_k =0 will be added to the end of the Q)
2:  sort containers in C_R in non-increasing order of h_ij
3:  while Q is not empty and C_R is not empty:
4:      t = Q.offer()              //get the first task from task queue
        assigned = false
5:      for each container c in C_R
6:          if t.L_k =0 then          //no local task anymore
7:              assign t to c
                assigned = true
8:              remove c from C_R
9:              break
10:         end if
11:         if container c has t's input data then // local task
12:             assign t to c         //assign local task
                assigned = true
13:             remove c from C_R
14:             break
15:         end if
16:     end for
        if (assigned = false) then
            put t to the end of the queue and set t.L_k =0    // since t's local container is already be occupied by another task.
17: end while
```

After a map task is dispatched, it will run the process described in Table 6.1.

6.3 Evaluation

To evaluate our scheduler, we create a hybrid cluster which has two types of nodes:

GPU nodes and CPU nodes. There are 2 GPU nodes, one has 112 CUDA cores (Geforce 9800-gt), the other has 16 CUDA cores (Geforce 210). We choose two types applications in our experiment. One includes MapReduce job that can employs GPU (aka. GPU job). The other is a general MapReduce job that uses CPU (aka. CPU job). In Table 6.5, there are specifications about the hybrid MapReduce cluster which has 2 GPU servers (each server has one GPU card) and 6 multi-core CPU servers.

Table 6.5 Experimental Environment

Nodes	Quantity	Hardware
Master node	1	2 single-core 2.2GHz Opteron-248 CPUs, 8GB RAM, 1Gbps Ethernet
GPU server Type I	1	2 single-core 2.2GHz Opteron-275 CPUs, 4GB RAM, Geforce 9800-gt GPU 512M RAM, 1 Gbps Ethernet
GPU server Type II	1	2 single-core 2.2GHz Opteron-275 CPUs, 4GB RAM, Geforce 210 GPU, 1Gbps Ethernet
CPU server	6	4 dual-core 2.2GHz Opteron-248 2.2G CPUs, 8GB RAM, 1Gbps Ethernet

To measure the energy consumption, we use a kill-a-watt meter [148] that connects to the cluster power outlet. It can measure the total energy consumption for the whole cluster.

6.3.1. Workload

We employ Facebook's workload [40] that has been used in our Chapter 4 and Chapter 5 to evaluate our energy-efficient scheduler against the Hadoop default FIFO scheduler.

Table 6.6 Workload I [20]

Bin	#Maps	%Jobs at Facebook	#Maps in Benchmark	# of jobs in Benchmark
1	1	39%	1	38
2	2	16%	2	16
3	3-20	14%	10	14
4	21-60	9%	50	8
5	61-150	6%	100	6
6	151-300	6%	200	6
7	301-500	4%	400	4
8	501-1,500	4%	800	4
9	>1,501	3%	4,800	4

Table 6.6 contains totally about 100 jobs. However, we have a relatively smaller cluster comparing with Facebook's production ones. We take the first 6 rows which covers about 87% of total jobs. Since the number of reduce tasks is not provided in their paper [40], we accordingly add reduce tasks for each category based on the number of map tasks. Basically, the number of reduce tasks is smaller than the number of map tasks. At the same time, to make it more general, we randomly pick the number of reduce tasks within a given interval for each category.

Table 6.7 Workload Configuration (in terms of number of map and reduce)

Bin	#Maps	#Reduces
1	1	1
2	2	[1,2]
3	10	[5,10]
4	50	[10,20]
5	100	[20,30]
6	200	30

This workload submission takes about 24 minutes and contains 87 jobs. The inter-arrival time follows the Poisson distribution with expectation of 14 seconds. Data accessing pattern is in the *zipf* distribution (skew = 1). Job size in the workload follows a Gaussian distribution. We mix two types of jobs: loadgen (CPU job) and MD simulation (GPU job) using MapReduce. For loadgen, each map task will take a data block as input. For MD simulation, its input data size is about 60KB. Two types of jobs are submitted randomly following the uniform distribution which means the number of jobs from each type is nearly the same (one is 44, the other is 45).

6.3.2. Energy Efficiency Profiling

Since we need to know each node's energy efficiency before scheduling any job, we did profiling of each node's energy efficiency for the two types of jobs.

For each job type, we run a single map task job on each container, measure the energy consumption during the job execution. We set the time interval as half hour (1,800 seconds), which guarantees the job can finish. Data is demonstrated in Table 6.8.

Table 6.8 Energy consumption of MD simulation job (1 map) on different types of nodes (1800 seconds sampling interval)

Node Type	Only use CPU	CPU + GPU
CPU node	0.1446 kwh	N/A
GPU node Type I	0.061 kwh	0.05858 kwh
GPU node Type II	0.0533 kwh	0.05185 kwh

We can see that using a GPU node can save energy for running GPU jobs. To make the result more intuitive, we employ the following method to obtain the energy efficiency factor of a node. For a given type of MapReduce job, the energy efficiency factor is the reciprocal of the energy consumption of this job on a server (i.e., the total energy minus the idle energy). We also normalize CPU node energy efficiency factor as 1 and get all other nodes' energy efficiency factor proportionally. For example, the CPU node's total energy consumption is 0.1446 kwh in the half hour sampling interval. To obtain the energy used by the MD simulation job, this 0.1446 kwh value should deduct the idle energy consumption in the sampling interval (235w*1,800s = 0.1175 kwh). We can see that only 0.0269 kwh is used for running a single task MD job on the CPU server. Since we take CPU node's energy efficient factor as 1, other nodes' energy efficient factor should divide 0.0269 kwh. We get Table 6.9.

Table 6.9 Energy efficiency factor for MD simulation

Node Type	Only use CPU	CPU + GPU
CPU node	1	N/A
GPU node Type I	1.23	1.41
GPU node Type II	1.4	1.52

In Table 6.10, we use data to demonstrate that it is energy efficient to run CPU job on CPU node. We also observed that the energy efficiency factor for GPU and CPU nodes are close if they both use CPU. This is because loadgen is a pure I/O job. Additionally, for type I GPU and type II GPU nodes, they are using the same CPU. Then, their energy efficiency factors are even closer.

Table 6.10 Energy efficiency factor for loadgen

Node Type	Only use CPU	CPU + GPU
CPU node	1	N/A
GPU node Type I	1.04	N/A
GPU node Type II	1.05	N/A

6.3.3. Experiment Results

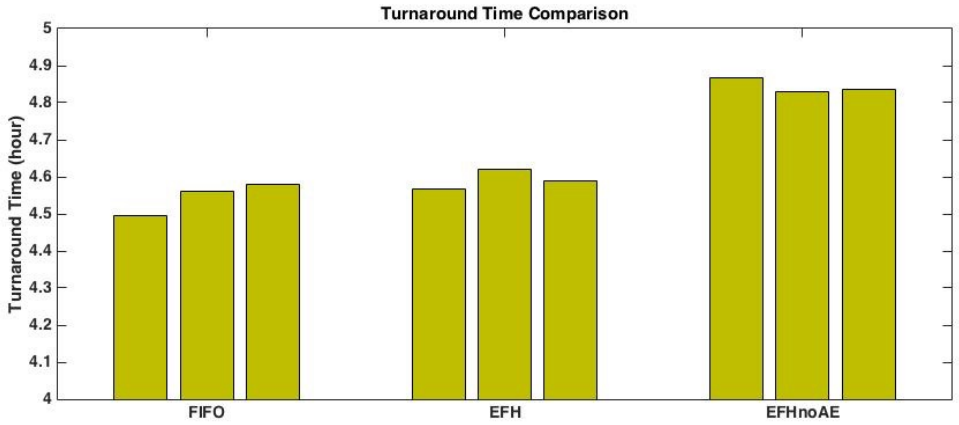
After having the energy efficiency factor of each application type, we start our final evaluation. The experiment time interval *T* is configured as 6.5 hours. Each experiment will be executed 3 times and we take average as final result.

Three metrics are introduced in this work:

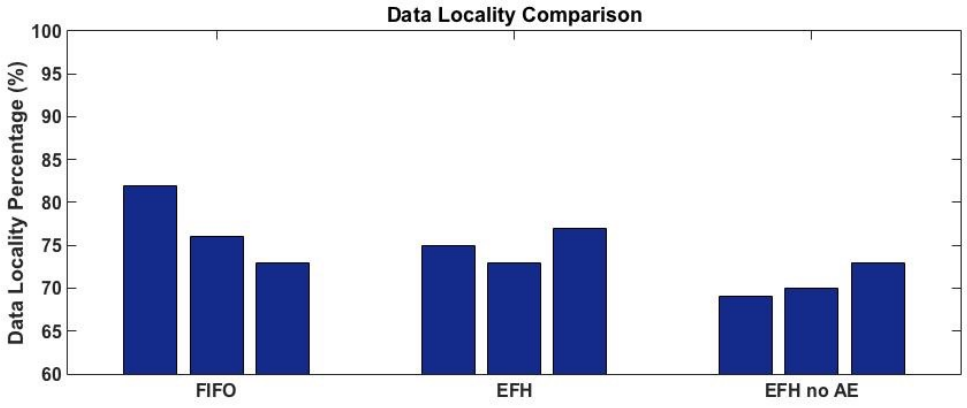
1. Workload turnaround time s the time interval between the first job arrival and the last job completion. This value is the smaller, the better.
2. Energy consumption is the total energy consumption during the given experiment time interval which is 6.5 hours. This value is the smaller, the better.
3. Data locality ratio is the ratio of the number of local map tasks divided by the number of all map tasks in the whole workload. We employ this metric to evaluate

the data locality performance of different schedulers. This value is the higher, the better, up to 100%.

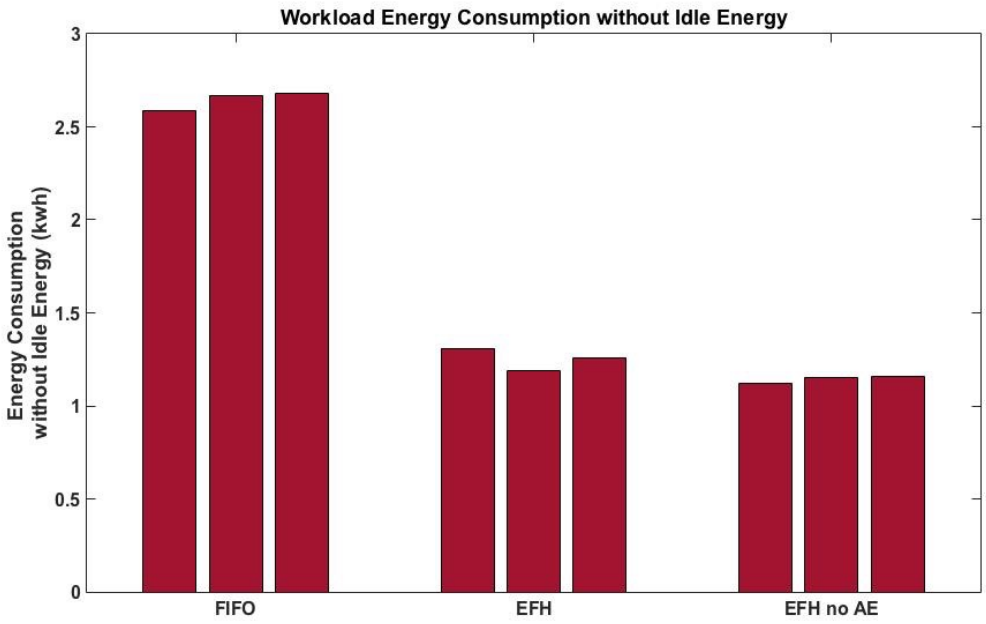
We compared three schedulers: FIFO, our scheduler (aka. Energy Efficient Hybrid, EFH for short), and EFH without adaptive execution (EFH-w/o-AE for short). The FIFO scheduler is the default scheduler of YARN. To avoid randomness, we run each experiment 3 times and take the average as the final result



a. **Turnaround time**



b. Data Locality Percentage



c. Energy Consumption

Figure 6.1. Turnaround time, data locality, and energy consumption for three schedulers

In Figure 6.1, we demonstrate all our results of three metrics. Each bar means one sampling point for a given scheduler. Each scheduler we get 3 times run and take the average as the final result. In Table 6.11, we demonstrate the final results.

6.3.3.1 FIFO vs. EFH w/o adaptive execution

We can see from Table 6.11, the EFH-w/o-AE scheduler consumes about 10% less energy in comparison to the FIFO scheduler which has no power management policy. For data centers that pays millions of dollars power bill, using our EFH scheduler can save about hundreds of thousands of dollars. We expect more energy saving if we have more GPU nodes or computation intensive jobs in the workload. We take 3 times run average

and also show all three times run result in parentheses.

Table 6.11 EFH schedulers without Adaptive Execution comparing with FIFO scheduler

Scheduler	Turnaround Time (hour)	Energy (kwh)	Energy no idle (kwh)	Data Locality Ratio (%)
FIFO	4.54 (4.49,4.56,4.57)	13.19 (13.13,13.22,13.21)	2.61(2.59, 2.57, 2.68)	77 (76,73,82)
EFH-w/o-AE	4.84 (4.87,4.83,4.83)	11.75 (11.66,11.73,11.87)	1.17 (1.15, 1.18, 1.19	71 (69,70,73)

The difference of data locality ratio between FIFO and EFH-w/o-AE scheduler is about 6%. This difference is not significant because our EFH-w/o-AE scheduler not only optimizes the energy consumption but also considers data locality.

For the workload turnaround time, the EFH-w/o-AE scheduler runs about 6.6% longer. It is as expected since we allow MapReduce jobs to wait for the best energy efficient resources. However, we have a 2-queue mechanism to prevent a job from starvation. At the same time, the data locality does not heavily affect the turnaround time since our GPU application's input data is relatively small (60KB). Even as the number of non-local map tasks increase, it did not cause too much network traffic or delay for the task execution.

6.3.3.2 EFH with and without adaptive execution

Table 6.12 EFH schedulers with and without adaptive execution

Scheduler	Turnaround Time (hour)	Energy (kwh)	Energy no idle (kwh)	Data Locality Ratio (%)
EFH-w/o-AE	4.84 (4.87,4.83,4.83)	11.75 (11.66,11.73,11.87)	1.17 (1.15, 1.18, 1.19	71 (69,70,73)
EFH	4.59 (4.57, 4.62,4.59)	11.78 (11.85, 11.69,11.80)	1.2 (1.26,1.14, 1.21)	75 (75,77,73)
IDEAL	16.73 (16.82, 17.14,16.23)	43.96(44.05,44.37, 43.46)	0.63 (0.65,0.68,0.55)	9.6 (11,10,8)

In Table 6.12, we can see the adaptive execution did contribute to the workload turnaround time. It gives an improvement of 5.2%. However, there is no free lunch, EFH scheduler consumes 1.9% more energy than EFH-w/o-AE. It allows some "long waiting" GPU jobs to run on a CPU node and trades energy consumption for time. The data locality ratio difference between these two schedulers is 4%. Since we have more CPU nodes than GPU nodes, it is possible that the CPU nodes hold more input data. Then, it inclines to achieve a higher data locality ratio when GPU jobs are allowed to run on CPU nodes.

6.3.3.3 Ideal Energy Consumption

In Figure 6.2, we add ideal energy consumption which is to run all map tasks on the node which is the most energy efficient without considering the cluster throughput. For example, if we have 10 map tasks that runs on node A is more energy efficient than running on node B, we will not allow any map task run on node B. In this way, we can get the ideal energy consumption for running a given workload on a given cluster. However, it may cause significant throughput degradation. However, we have this ideal result is to

show the upper bound. To make it comparable to our previous data, we deduct all idle energy consumption for three candidate schedulers.

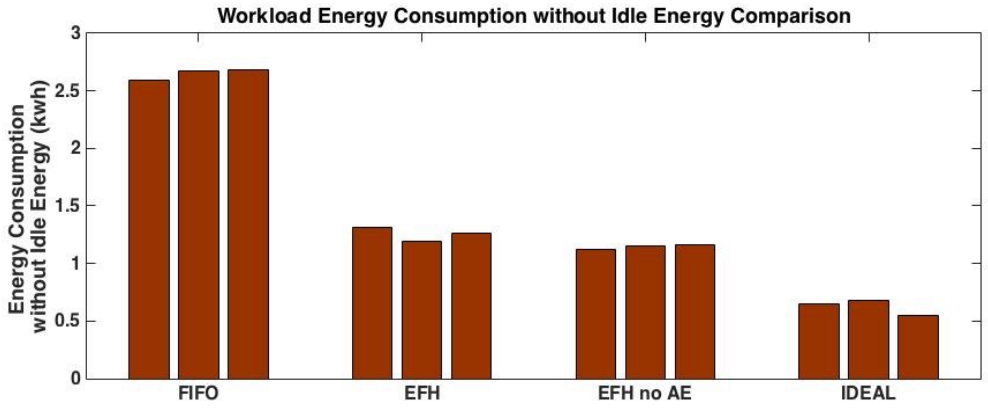


Figure 6.2 Energy Consumption with IDEAL Energy Run (no idle energy)

In Figure 6.3, we see the execution time is about 3 times longer than other three schedulers. This is as expected because we only run map tasks on the most energy efficient nodes. Other servers are idle.

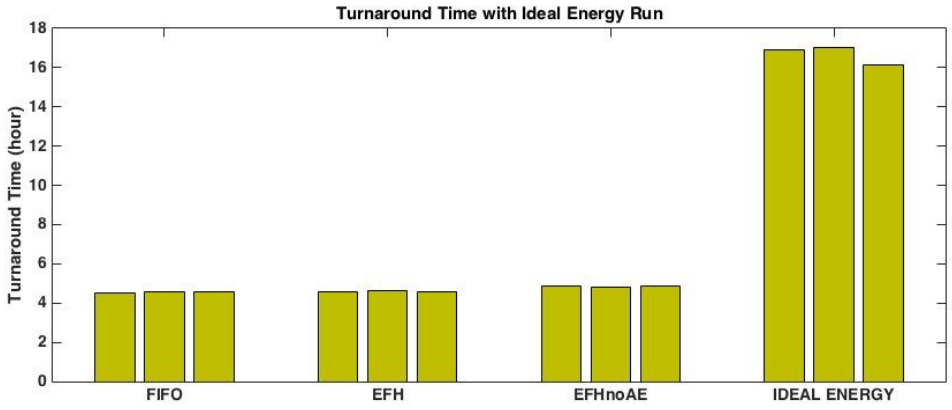


Figure 6. 3 Turnaround Time with IDEAL Energy Run

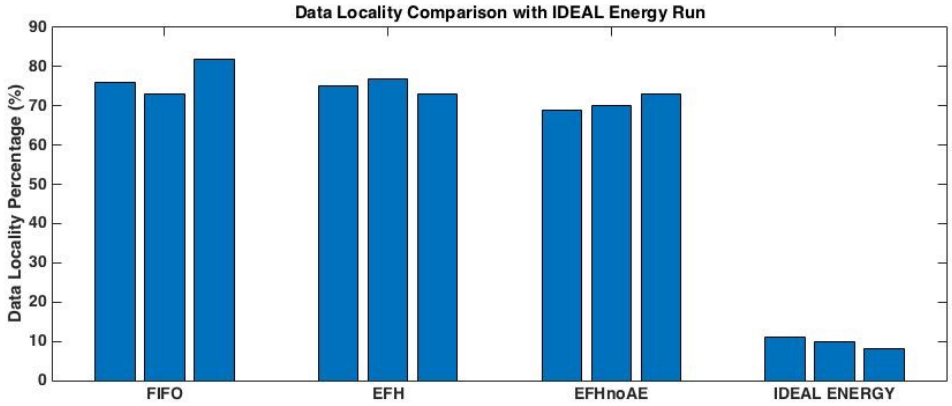


Figure 6. 4 Data Locality with IDEAL Energy Run

In Figure 6.4, we can see the data locality in IDEAL Energy group is about 10% . It is significant degradation comparing with other three schedulers. Since we only run map tasks on the most energy efficient nodes, this is inevitable.

6.3.3.4 Statement of Result Statistics

In order to further analyze our experiment results, we did some statistical comparison between EFH, FIFO, and EFHnoAE schedulers. It includes: p-value with t test, confidence interval, and standard error estimation.

First of all, we use p-value[154] to show the asymptotic significance of experimental data. In Table 6.13, we show p-value of energy consumption, data locality, and turnaround time. According to the p-value definition, in our evaluation, if $p < 0.05$, it means there is high probability that two sets of data come from different distribution. For turnaround time, FIFO and EFHnoAE have significant difference, as well as EFH and EFHnoAE. But FIFO and EFH are close. For data locality, the significances for 3 candidates are small. For energy consumption, we can see FIFO and EFH has significant diffe-

rence. At the same time, FIFO and EFHnoAE are significantly different.

Table 6. 13 P-value with significance level 0.05

Scheduler	Turnaround Time	Data Locality	Energy Consumption
FIFO vs. EFH	0.0793	0.65	0.0014
FIFO vs. EFHnoAE	0.0129	0.0696	0.0021
EFH vs. EFHnoAE	0.0105	0.1859	0.4266

On the other hand, we provide confidence intervals for three scheduler experiment results. Results are shown in Table 6.14. With 95% confidence interval, we can see, in the turnaround time, EFH data variance is smaller than FIFO and EFHnoAE. For data locality, FIFO data has the larger variance. For energy consumption, EFHnoAE has the largest variance.

Table 6. 14 Confidence intervals with 95% confidence

Scheduler	Turnaround Time (hour)	Data Locality (%)	Energy Consumption (kwh)
FIFO	4.54 (+/-0.1)	77 (+/-11)	13.19 (+/-0.13)
EFH	4.59 (+/-0.06)	75 (+/-5)	11.79 (+/-0.13)
EFH no AE	4.84 (+/-0.1)	70.67 (+/-5)	11.75 (+/-0.28)

In the end, we provide the error estimation for our three scheduler results. They are in Table 6.15. The standard error for our experiment data are relatively small except data locality.

Table 6. 15 Standard Error Estimation

Scheduler	Turnaround Time (hour)	Data Locality (%)	Energy Consumption (kwh)
FIFO	0.02	2.16	0.02
EFH	0.01	0.94	0.03
EFH no AE	0.01	0.98	0.05

CHAPTER 7. CONCLUSION AND FUTURE WORK

In this dissertation, we studied, designed, developed, and evaluated three schedulers for the Hadoop MapReduce framework step by step to approach our proposed target: to provide MapReduce applications with low cost and energy consumption through the development of scheduling theory and algorithms, energy models, and energy-aware resource management [42,74-77]. (refer all my previous publications)

First of all, we investigate Hadoop MapReduce framework's data locality mechanism and develop a matchmaking scheduling algorithm for improving the data locality of MapReduce applications. Evaluation using a Facebook workload shows our scheduler can adaptively achieve a high data locality ratio and a shorter map task response time comparing with the delay scheduler and the Hadoop default scheduler.

Secondly, a real-time scheduling algorithm has been developed for MapReduce applications that require QoS and run in heterogeneous Hadoop MapReduce clusters. A mathematical proof has been provided as well as a real cluster evaluation. Both confirmed our real-time scheduler can achieve higher cluster utilization without deadline missing comparing with deadline constraint scheduler.

Last but not least, we proposed an energy efficient scheduler for Hadoop YARN to resolve a multi-constraint optimization problem: saving cluster power consumption, satisfying MapReduce applications increasing demands on computation power, considering data locality, and avoiding performance degradation. In this work, we proposed our two

levels scheduling algorithm. To evaluate our scheduler, we build a hybrid Hadoop cluster which has two types of computing nodes: GPU nodes and CPU nodes. Comparing with the Hadoop YARN default scheduler, our algorithm can save about 10% energy without significant performance degradation. Since Hadoop YARN is a general resource scheduler, our algorithm can also benefit not only MapReduce applications but also other frameworks like Apache TEZ[149], Apache Spark[150], etc.

With the increasing growth of public cloud applications, we will focus our future work on Big Data clusters resource scheduling in the cloud environment with the help of machine learning algorithms. Based on our previous study in Big Data framework and the cloud environment, our future plan is to develop an intelligent scheduler. It is able to anticipate the workload peak and automatically scale out or scale in resources by learning from the historical workloads. For example, the Hadoop cluster may encounter workload burst in some special holiday like Thanksgiving, Christmas, etc. A smart scheduler should be able to request more resources before the burst happens and release idle resources after the rush hours. The elasticity of the cloud provides a possible infrastructure for the smart scheduler to realize this feature adaptively. However, the cloud environment is more complicated than dedicated clusters, it needs more investigation and research work to be done in this area. We will carry forward our learned knowledge, research skills, and experiences to forge ahead in the future.

REFERENCES

1. Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.
2. Hadoop http://www.hadoop.com
3. Big Data, https://en.wikipedia.org/wiki/Big_data
4. Kong, Fansheng, and Xiaola Lin. "The method and application of big data mining for mobile trajectory of taxi based on MapReduce." Cluster Computing (2018): 1-8.
5. Carcillo, Fabrizio, et al. "Scarff: a scalable framework for streaming credit card fraud detection with spark." Information fusion 41 (2018): 182-194.
6. Hosseini, Behrooz, and Kourosh Kiani. "FWCMR: A scalable and robust fuzzy weighted clustering based on MapReduce with application to microarray gene expression." Expert Systems with Applications 91 (2018): 198-210.
7. Zhu, Fubao, et al. "A Classification Algorithm of CART Decision Tree based on MapReduce Attribute Weights." International Journal of Performability Engineering 14.1 (2018): 17.
8. M.C. Schatz, “BlastReduce: high performance short read mapping with MapReduce”. http://www.cbcb.umd.edu/software/blastreduce/
9. M. Zaharia Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, “Improving MapReduce performance in heterogeneous environments”, in: Proc. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, San Diego,USA, Dec. 2008.
10. Kong, Fansheng, and Xiaola Lin. "The method and application of big data mining for mobile trajectory of taxi based on MapReduce." Cluster Computing (2018): 1-8.
11. Polo, Jorda, et al. "Performance-driven task co-scheduling for mapreduce environments." Network Operations and Management Symposium (NOMS), 2010 IEEE. IEEE, 2010.
12. Xicheng Dong, Ying Wang, Huaming Liao “Scheduling Mixed Real-time and Non-real-time Applications in MapReduce Environment”. In the proceeding of 17th International Conference on Parallel and Distributed Systems. 2011, pp. 9 – 16
13. Marco A. S. Netto and Rajkumar Buyya. "Offer-based scheduling of deadline-constrained bag-of- tasks applications for utility computing systems". In Proceedings of the 18th International Heterogeneity in Computing Workshop, in conjunction with the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS’09), Roma, Italy, May 2009.
14. Verma, Abhishek, Ludmila Cherkasova, and Roy H. Campbell. "ARIA: automatic resource inference and allocation for mapreduce environments." Proceedings of the 8th ACM international conference on Autonomic computing. ACM, 2011.
15. Phan, Linh TX, et al. "An empirical analysis of scheduling techniques for real-time cloud-based data processing." Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on. IEEE, 2011.
16. Liu, Li, et al. "Preemptive Hadoop Jobs Scheduling under a Deadline." Semantics, Knowledge and Grids (SKG), 2012 Eighth International Conference on. IEEE, 2012.
17. K. Kc and K. Anyanwu, “Scheduling hadoop jobs to meet deadlines,” in 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2010, pp. 388 –392.
18. D. Luebke, and M. Harris. "General-purpose computation on graphics hardware." Workshop, SIGGRAPH. 2004.
19. J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens, “Multi-gpu volume rendering using mapreduce,” 1st International Workshop on MapReduce and its Applications, June 2010.
20. Jeff A. Stuart and John D. Owens. "Multi-GPU MapReduce on GPU Clusters". In IPDPS, 2011.
21. F. Ji and X. Ma. "Using Shared Memory to Accelerate Mapreduce on Graphics Processing Units". In IPDPS ’11, pages 805–816, 2011
22. Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. "Hybrid map task scheduling for gpu-based heterogeneous clusters." Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on. IEEE, 2010.

23. Sundaresan Venkatasubramanian and Richard W. Vuduc. "Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems". In ICS '09: Proceedings of the 23rd international conference on Supercomputing, pages 244–255, New York, NY, USA, 2009. ACM
24. Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. "A Map Reduce Framework for Programming Graphics Processors". In Third Workshop on Software Tools for MultiCore Systems (STMCS), 2008
25. Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. "MapCG: Writing Parallel Program Portable between CPU and GPU". In PACT, pages 217–226, 2010.
26. L. Chen, and Gagan Agrawal. "Optimizing MapReduce for GPUs with effective shared memory usage." Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing. ACM, 2012.
27. Stuart, Jeff A., and John D. Owens. "Multi-GPU MapReduce on GPU clusters." Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International. IEEE, 2011.
28. Chen, Cen, et al. "Gflink: An in-memory computing architecture on heterogeneous CPU-GPU clusters for big data." IEEE Transactions on Parallel and Distributed Systems (2018).
29. Feng, Jiaying, Xiaowang Zhang, and Zhiyong Feng. "MapSQ: A MapReduce-based Framework for SPARQL Queries on GPU." arXiv preprint arXiv:1702.03484 (2017).
30. Jiang, Hai, et al. "Scaling up MapReduce-based big data processing on multi-GPU systems." Cluster Computing 18.1 (2015): 369-383.
31. Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. "Multi-core real-time scheduling for generalized parallel task models." Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd (pp. 217-226). IEEE.
32. Li, Shen, Shaohan Hu, and Tarek Abdelzaher. "The packing server for real-time scheduling of mapreduce workflows." Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE. IEEE, 2015.
33. GPGPU http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
34. The next generation Apache Hadoop. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
35. Yao, Yi, et al. "Haste: Hadoop yarn scheduling based on task-dependency and resource-demand." Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on. IEEE, 2014.
36. Lin, Jia-Chun, et al. "ABS-YARN: A formal framework for modeling Hadoop YARN clusters." the International Conference on Fundamental Approaches to Software Engineering. Springer, Berlin Heidelberg, 2016.
37. CUDA http://developer.NVIDIA.com/cuda.
38. D.Luebke. "CUDA: Scalable parallel programming for high performance scientific computing." Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on. IEEE, 2008.
39. J. Nickolls, I. Buck, M. Garland, and K. Skadron,"Scalable parallel programming with CUDA," ACM Queue, pp. 40–53, Mar./Apr. 2008
40. Zaharia, Matei, et al. "Improving MapReduce performance in heterogeneous environments." Osdi. Vol. 8. No. 4. 2008.
41. Chen, Quan, et al. "Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment." Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on. IEEE, 2010.
42. Sun, Xiaoyu, Chen He, and Ying Lu. "ESAMR: An Enhanced Self-Adaptive MapReduce Scheduling Algorithm." Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on. IEEE, 2012.
43. Linux Container. http://lxc.sourceforge.net/
44. Capacity Scheduler, https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html
45. Zaharia, Matei. "Job scheduling with the fair and capacity schedulers." Hadoop Summit 9 (2009).
46. Lable Scheduling, https://hadoop.apache.org/docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/NodeLabel.html

47. YARN-6223, https://issues.apache.org/jira/browse/YARN-6223
48. NVIDIA, http://www.nvidia.com
49. Y. Yan, M. Grossman, and V. Sarkar, “Jcuda: A programmer-friendly interface for accelerating java programs with cuda,” Lecture Notes in Computer Sciences, vol. 5704 (2009), pp. 887–899, 2009
50. GPUs are driving energy efficiency across the computing industry, from phones to super computers. http://www.nvidia.com/object/gcr-energy-efficiency.html
51. Xie, Qiaomin, et al. "Pandas: Robust locality-aware scheduling with stochastic delay optimality." IEEE/ACM Transactions on Networking (TON) 25.2 (2017): 662-675.
52. Isard, Michael, et al. "Dryad: distributed data-parallel programs from sequential building blocks." ACM SIGOPS Operating Systems Review 41.3 (2007): 59-72.
53. Polato, Ivanilton, et al. "A comprehensive view of Hadoop research—A systematic literature review." Journal of Network and Computer Applications 46 (2014): 1-25.
54. Althebyan, Qutaibah, et al. "Evaluating map reduce tasks scheduling algorithms over cloud computing infrastructure." Concurrency and Computation: Practice and Experience27.18 (2015): 5686-5699.
55. Xie, Qiaomin, Ali Yekkehkhany, and Yi Lu. "Scheduling with multi-level data locality: Throughput and heavy-traffic optimality." INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE. IEEE, 2016.
56. Gaur, Manisha, Bhawna Minocha, and Sunil Kumar Muttoo. "A Study of Factors Affecting MapReduce Scheduling." Big Data Analytics. Springer, Singapore, 2018. 275-281.
57. Ekanayake, Jaliya, Shrideep Pallickara, and Geoffrey Fox. "Mapreduce for data intensive scientific analyses." eScience, 2008. eScience'08. IEEE Fourth International Conference on. IEEE, 2008.
58. He, Bingsheng, et al. "Mars: a MapReduce framework on graphics processors." Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 2008.
59. Shan, Yi, et al. "Fpmr: Mapreduce framework on fpga." Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2010.
60. Yang, Hung-chih, et al. "Map-reduce-merge: simplified relational data processing on large clusters." Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007.
61. De Kruijf, Marc, and Karthikeyan Sankaralingam. "Mapreduce for the cell broadband engine architecture." IBM Journal of Research and Development 53.5 (2009): 10-1.
62. Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. “Phoenix Rebirth: Scalable mapReduce on a Large-Scale Shared-Memory System” in IISSWC, pg 198-207,2009
63. Computing Grid. http://www.e-sciencecity.org/EN/gridcafe/what-is-the-grid.html
64. T.Sandholm, K. Lai, “Dynamic Proportional Share Scheduling in Hadoop”, in Proceedings of the 15th workshop on job scheduling strategies for parallel processing, pg 110-131, 2010
65. Kambatla, Karthik, Abhinav Pathak, and Himabindu Pucha. "Towards optimizing hadoop provisioning in the cloud." Proc. of the First Workshop on Hot Topics in Cloud Computing. 2009.
66. Faraz Ahmad, et al. “Tarazu: Optimizing MapReduce on Heterogeneous Clusters”, ASPLOS’12 March 3, 2012, London, England, UK
67. Hong Mao, et al. “A Load-Driven Task Scheduler with Adaptive DSC for MapReduce”, GreenCom 2011.
68. Bu, Yingyi, et al. "HaLoop: Efficient iterative data processing on large clusters." Proceedings of the VLDB Endowment 3.1-2 (2010): 285-296.
69. Li, Qiuhong, et al. "LI-MR: A Local Iteration Map/Reduce Model and Its Application to Mine Community Structure in Large-Scale Networks." Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on. IEEE, 2011.
70. Zhang, Yanfeng, et al. "imapreduce: A distributed computing framework for iterative computation." Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on. IEEE, 2011.
71. Cascading http://www.cascading.org/
72. Eslam Elnikety, Tamer Elsayed, Hany E. Ramadan, “iHadoop: Asynchronous Iterations for MapReduce”, CloudCom 2011, Athens, Greece

73. Condie, Tyson, et al. "MapReduce online." Proceedings of the 7th USENIX conference on Networked systems design and implementation. 2010.
74. Kune, Raghavendra, et al. "Genetic algorithm based data-aware group scheduling for Big Data clouds." Big Data Computing (BDC), 2014 IEEE/ACM International Symposium on. IEEE, 2014.
75. Li, Runhui, and Patrick PC Lee. "Making mapreduce scheduling effective in erasure-coded storage clusters." Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on. IEEE, 2015.
76. Shen, Haiying, et al. "Probabilistic network-aware task placement for mapreduce scheduling." Cluster Computing (CLUSTER), 2016 IEEE International Conference on. IEEE, 2016.
77. Hashem, Ibrahim Abaker Targio, et al. "Multi-objective scheduling of MapReduce jobs in big data processing." Multimedia Tools and Applications (2017): 1-16.
78. He, Chen, Ying Lu, and David Swanson. "Matchmaking: A new mapreduce scheduling technique." Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on. IEEE, 2011.
79. He, Chen, Ying Lu, and David Swanson. "Real-time scheduling in mapreduce clusters." High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on. IEEE, 2013.
80. He, Chen. "Molecular dynamics simulation based on hadoop mapreduce." (2011).
81. M. Mustafa Rafique, Benjamin Rose, Ali Raza Butt, and Dimitrios S. Nikolopoulos. CellMR: A Framework for Supporting Mapreduce on Asymmetric Cell-Based Clusters. In IPDPS, pages 1–12, 2009.
82. Marwa Elteir, Heshan Lin, and Wu-chun Feng. StreamMR: An Optimized MapReduce Framework for AMD GPUs. In ICPADS ’11, Tainan, Taiwan, December 2011
83. Chen He, Derek Weitzel, Ying Lu, David Swanson, “HOG: Distributed Hadoop MapReduce on the Grid”, in the proceeding of SC12, 5th MTAGS Workshop
84. Sangwon Seo, Ingook Jang, et al., “HPMR: Prefetching and Pre-Shuffling in Shared MapReduce Computation Environment.” IEEE International Conference on Cluster Computing and Workshops, 2009
85. X. Zhang, Z. Zhong, S. Feng, B. Tu, J. Fan, “Improving Data Locality of MapReduce by Scheduling in Homogeneous Computing Environments”, in 9th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 120-126, 2011
86. M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Job scheduling for multi-user mapreduce clusters,” EECS Department, University of California, Berkeley, Tech. Rep., Apr 2009.
87. Dhall, Sudarshan K., and C. L. Liu. "On a real-time scheduling problem." Operations Research 26.1 (1978): 127-140.
88. Jensen, E. Douglas, C. Douglass Locke, and Hideyuki Tokuda. "A time-driven scheduling model for real-time operating systems." Proceedings of the 6th IEEE Real-Time Systems Symposium. 1985.
89. Perkins, James R., and P. R. Kumark. "Stable, distributed, real-time scheduling of flexible manufacturing/assembly/diassembly systems." Automatic Control, IEEE Transactions on 34.2 (1989): 139-148.
90. Audsley, Neil C., et al. "‘Hard Real-Time Scheduling: The Deadline Monotonic Approach." Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software. 1991.
91. Van Tilborg, Andre M., and Gary M. Koob. Foundations of real-time computing: Scheduling and resource management. Vol. 141. Springer, 1991.
92. Hyman, Jay M., Aurel A. Lazar, and Giovanni Pacifici. "Real-time scheduling with quality of service constraints." Selected Areas in Communications, IEEE Journal on 9.7 (1991): 1052-1063.
93. Sha, Lui, and John B. Goodenough. "Real-time scheduling theory and Ada." Mission-Critical Operating Systems (1992): 294-319.
94. Stankovic, John A., et al. "Implications of classical scheduling results for real-time systems." Computer 28.6 (1995): 16-25.

95. Garvey, Alan J., and Victor R. Lesser. "Design-to-time real-time scheduling." Systems, Man and Cybernetics, IEEE Transactions on 23.6 (1993): 1491-1502.
96. Stankovic, John A., et al. "The case for feedback control real-time scheduling." Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on. IEEE, 1999.
97. Shakkottai, Sanjay, and Alexander L. Stolyar. Scheduling algorithms for a mixture of real-time and non-real-time data in HDR. Bell Laboratories, Lucent Technologies, 2000.
98. Lu, Chenyang, et al. "Feedback control real-time scheduling: Framework, modeling, and algorithms*." Real-Time Systems 23.1 (2002): 85-126.
99. Scott A. Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In Proceedings of the 24th IEEE International Real-Time Systems Symposium, pages 396–409, December 2003. Cancun, Mexico.
100. Sha, Lui, et al. "Real time scheduling theory: A historical perspective." Real-time systems 28.2 (2004): 101-155.
101. C. L. Liu and James, W, Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment”, J. ACM, 20:46-61. Jan 1973
102. K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. IEEE Transactions on Computers, 38(8):1110–1123, 1989. ISSN 0018-9340.
103. M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. IEEE Trans. Softw. Eng., 15(12):1497–1506, 1989. ISSN 0098-5589.
104. Krithi Ramamritham, John A. Stankovic, and Perng fei Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. IEEE Trans. on Parallel and Distributed Systems, 1(2): 184–194, April 1990.
105. Davender Babbar and Phillip Krueger. On-line hard real-time scheduling of parallel tasks on partitionable multiprocessors. In ICPP, pages 29–38, 1994.
106. G. Manimaran and C. S. R. Murthy. An efficient dynamic scheduling algorithm for multiprocessor real-time systems. IEEE Trans. on Parallel and Distributed Systems, 9(3):312–319, 1998. URL citeseer.ist.psu.edu/manimaran98efficient.html.
107. David Steere, Ashvin Goel, Joshua Gruenberg, Dylan Mcnamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling, 1999.
108. Xiao Qin and Hong Jiang. Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems. In Proc. of 30th International Conference on Parallel Processing, pages 113–122, Valencia, Spain, September 2001.
109. John A. Stankovic, Tian He, Tarek Adbelzaher, Mike Marley, Gang Tao, Sang Son, and Chenyang Lu. Feedback control scheduling in distributed real-time systems*. In In IEEE Real-Time Systems Symposium, pages 59–72, 2001.
110. Lichen Zhang. Scheduling algorithm for real-time applications in grid environment. In Proc.of IEEE International Conference on Systems, Man and Cybernetics, volume 5, page 6 pp., Hammamet, Tunisia, October 2002.
111. Reda A. Ammar and Abdulrahman Alhamdan. Scheduling real time parallel structure on cluster computing. In Proc. of 7th IEEE International Symposium on Computers and Communications, pages 69–74, Taormina, Italy, July 2002.
112. Tyagi, Rinki, and Santosh Kumar Gupta. "A Survey on Scheduling Algorithms for Parallel and Distributed Systems." Silicon Photonics & High Performance Computing. Springer, Singapore, 2018. 51-64.
113. Xie, Guoqi, et al. "High performance real-time scheduling of multiple mixed-criticality functions in heterogeneous distributed embedded systems." Journal of Systems Architecture 70 (2016): 3- 14.
114. Alghamdi, Mohammed I., et al. "Towards two-phase scheduling of real-time applications in distributed systems." Journal of Network and Computer Applications 84 (2017): 109-117.
115. Chen, Fangbing, Ji Liu, and Yuesheng Zhu. "A Real-Time Scheduling Strategy Based on Processing Framework of Hadoop." Big Data (BigData Congress), 2017 IEEE International Congress on. IEEE, 2017.
116. Thirunavukkarasu, Gokul Sidarth, and Ragil Krishna. "Scheduling Algorithm for Real-Time Embedded Control Systems using Arduino Board." KnE Engineering 2.2 (2017): 258-266.

117. Dantong Yu and Thomas G. Robertazzi. Divisible load scheduling for grid computing. In Proc.of IASTED International Conference on Parallel and Distributed Computing and Systems, Los Angeles, CA, November 2003.
118. Yao, Yi, et al. "Haste: Hadoop yarn scheduling based on task-dependency and resource-demand." Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on. IEEE, 2014.Lin, Jia-Chun, et al. "ABS-YARN: A formal framework for modeling Hadoop YARN clusters." International Conference on Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, 2016.
119. Yigitbasi, Nezih, et al. "Energy efficient scheduling of MapReduce workloads on heterogeneous clusters." Green Computing Middleware on Proceedings of the 2nd International Workshop. ACM, 2011.
120. Chen, Yanpei, et al. "Energy efficiency for large-scale mapreduce workloads with significant interactive analysis." Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012.
121. Fan, Zhe, et al. "GPU cluster for high performance computing." Proceedings of the 2004 ACM/IEEE conference on Supercomputing. IEEE Computer Society, 2004.
122. Owens, John. GPUs: Engines for future high-performance computing. CALIFORNIA UNIV DAVIS, 2004.
123. D. Luebke, and M. Harris. "General-purpose computation on graphics hardware." Workshop, SIGGRAPH. 2004.
124. M. Pharr, and R. Fernando. "GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation Author: Matt Pharr, Randi." (2005): 880.
125. D. Luebke. "CUDA: Scalable parallel programming for high-performance scientific c o mputing." Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on. IEEE, 2008.
126. J. Nickolls, I. Buck, M. Garland, and K. Skadron,"Scalable parallel programming with CUDA," ACM Queue, pp. 40–53, Mar./Apr. 2008
127. J. D.Owens, M. Houston, D. Luebke, S. Green, J. E.Stone, and J. C.Phillips, "GPU c o mputing," Proc IEEE, vol. 96, no. 5, pp. 879–899,2008.
128. R. Colby, R. Ramanan, P. Arun, B. Gary, and K. Christos, "Evaluating mapreduce for multicore and multiprocessor systems," in Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.
129. Lee, Seyong, Seung-Jai Min, and Rudolf Eigenmann. "OpenMP to GPGPU: a compiler framework for automatic translation and optimization." ACM Sigplan Notices 44.4 (2009): 101- 110.
130. Y. Yan, M. Grossman, and V. Sarkar, "Jcuda: A programmer-friendly interface for accelerating java programs with cuda," Lecture Notes in Computer Sciences, vol. 5704 (2009), pp. 887–899, 2009
131. B. Hong-tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He, "K-means on commodity gpus with cuda," Computer Science and Information Engineering, 2009 WRI World Congress, pp. 651–655, 2009.
132. Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin:exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 45–55, New York, NY, USA, 2009. ACM.
133. Vignesh T. Ravi, Wenjing Ma, Vignesh T. Ravi, and Gagan Agrawal. Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations'. In Proceedings of International Conference on Supercomputing (ICS), 2010.
134. Sundaresan Venkatasubramanian and Richard W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In ICS '09: Proceedings of the 23rd international conference on Supercomputing, pages 244–255, New York, NY,USA, 2009. ACM
135. Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A Map Reduce Framework for Programming Graphics Processors. In Third Workshop on Software Tools for MultiCore Systems (STMCS), 2008

136. Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: Writing Parallel Program Portable between CPU and GPU. In PACT, pages 217–226, 2010.
137. Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. "Hybrid map task scheduling for gpu-based heterogeneous clusters." Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on. IEEE, 2010.
138. J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens, "Multi-gpu volume rendering using mapreduce," 1st International Workshop on MapReduce and its Applications, June 2010.
139. F. Ji and X. Ma. Using Shared Memory to Accelerate Mapreduce on Graphics Processing Units. In IPDPS '11, pages 805–816, 2011
140. L. Chen, and Gagan Agrawal. "Optimizing MapReduce for GPUs with effective shared memory usage." Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing. ACM, 2012.
141. Qiao, Zhi, et al. "MR-Graph: a customizable GPU MapReduce." Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on. IEEE, 2015.
142. Liu, Lifeng, et al. "A-MapCG: An Adaptive MapReduce Framework for GPUs." Networking, Architecture, and Storage (NAS), 2017 International Conference on. IEEE, 2017.
143. Lang, Willis, and Jignesh M. Patel. "Energy management for mapreduce clusters." Proceedings of the VLDB Endowment 3.1-2 (2010): 129-139.
144. Wirtz, Thomas, and Rong Ge. "Improving MapReduce energy efficiency for computation intensive workloads." Green Computing Conference and Workshops (IGCC), 2011 International. IEEE, 2011.
145. Cardosa, Michael, et al. "Exploiting spatio-temporal tradeoffs for energy-aware MapReduce in the Cloud." Cloud Computing (CLOUD), 2011 IEEE International Conference on. IEEE, 2011.
146. Chou, Jerry, Jinoh Kim, and Doron Rotem. "Energy-aware scheduling in disk storage systems." Distributed Computing Systems (ICDCS), 2011 31st International Conference on. IEEE, 2011.
147. Yigitbasi, Nezih, et al. "Energy efficient scheduling of MapReduce workloads on heterogeneous clusters." Green Computing Middleware on Proceedings of the 2nd International Workshop. ACM, 2011.
148. Chen, Yanpei, et al. "Energy efficiency for large-scale mapreduce workloads with significant interactive analysis." Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012.
149. Ren, Da Qi. "Algorithm level power efficiency optimization for CPU–GPU processing element in data intensive SIMD/SPMD computing." Journal of Parallel and Distributed Computing 71.2 (2011): 245-253.
150. Liu, Wenjie, et al. "A waterfall model to achieve energy efficient tasks mapping for large scale gpu clusters." Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on. IEEE, 2011.
151. Huo, Hongpeng, et al. "An energy efficient task scheduling scheme for heterogeneous GPU-enhanced clusters." Systems and Informatics (ICSAI), 2012 International Conference on. IEEE, 2012,(pp.623-627)
152. Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin:exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 45–55, New York, NY, USA, 2009. ACM.
153. Kim, SungYe, et al. "Power efficient mapreduce workload acceleration using integrated-gpu." Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on. IEEE, 2015.
154. P-value, https://en.wikipedia.org/wiki/P-value